INTRODUCTION
TO
SCILAB

INRIA/ENPC

INRIA - Rocquencourt unit - Metalau project.
Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)
ENPC - Cité Descartes -Champs sur Marne- 77455 Marne la Vallée Cedex (France)
E-mail : scilab@inria.fr
Home page : www.scilab.org

Chapter 1

Introduction

1.1 What is Scilab

Developed at INRIA and ENPC, Scilab has been developed for engineering applications. It is freely distributed (see the copyright file) and the source code is available.

Scilab is made of three distinct parts: an interpreter, a set of libraries of Fortran and C routines linked with Scilab and a number of toolboxes written in Scilab language. The numerical libraries (which, strictly speaking, do not belong to Scilab but are interactively called by the interpreter) are of independent interest and most of them are available through the Web.

A key feature of the Scilab syntax is its ability to handle matrices: basic matrix manipulations such as concatenation, extraction or transpose are immediately performed as well as basic operations such as addition or multiplication. The syntax for manipulating matrices is mostly compatible with Matlab. Scilab also aims at handling more complex objects than numerical matrices. This is done in Scilab by manipulating structures which allows a natural symbolic representation of complicated objects such as transfer functions, linear systems or graphs (see Section 2.8).

Mathematical objects such as polynomials, polynomials matrices and rational polynomial matrices are also defined and the syntax used for manipulating these matrices is identical to that used for manipulating numerical vectors and matrices.

Scilab provides a variety of powerful primitives for the analysis of non-linear systems. Integration of explicit and implicit dynamic systems can be accomplished numerically. The `scicos` toolbox allows the graphic definition and simulation of complex interconnected hybrid systems. Documentation about `scicos` can be found at its the Web page scicos.org.

There exist numerical optimization facilities for non linear optimization (including non differentiable optimization), quadratic optimization and linear optimization (see also the contribution directory on the Scilab Web page).

Scilab has an open programming environment where the creation of functions and libraries of functions is completely in the hands of the user (see Chapter 3). Functions are recognized as data objects in Scilab and, thus, can be manipulated or created as other data objects. For example, functions can be defined inside Scilab and passed as input or output arguments of other functions.

In addition Scilab supports a character string data type which, in particular, allows the on-line creation of functions. Matrices of character strings are also manipulated with the same syntax as ordinary matrices.

Finally, Scilab is easily interfaced with Fortran or C subprograms. This allows use of standardized packages and libraries in the interpreted environment of Scilab.

The general philosophy of Scilab is to provide the following sort of computing environment:

- To have data types which are varied and flexible with a syntax which is natural and easy to use.
- To provide a reasonable set of primitives which serve as a basis for a wide variety of calculations.
- To have an open programming environment where new primitives are easily added.
- To support library development through "toolboxes" of functions devoted to specific applications (linear control, signal processing, network analysis, non-linear control, etc.)

The objective of this introduction manual is to give the user an idea of what Scilab can do. On line documentation about all functions is available (`help` command).

1.2 Software Organization

Scilab is divided into a set of directories. The main directory `SCIDIR` contains the following files: `scilab.star` (startup file), the copyright file `notice.tex`, and the `configure` files (see(1.3)). The main subdirectories are the following:

- `bin` is the directory of the executable files. The starting script `scilab` on Unix/Linux systems and `runscilab.exe` on Windows. The executable code of Scilab: `scilex` on Unix/Linux systems and `scilex.exe` on Windows are there. This directory also contains Shell scripts for managing or printing Postscript/LaTeX files produced by Scilab.
- `demos` is the directory of demos. This directory contains the codes corresponding to various demos. They are often useful for inspiring new users. The file `alldems.dem` is used by the "Demos" button. Most of plot commands are illustrated by simple demo examples. Note that running a graphic function without input parameter provides an example of use for this function (for instance `plot2d()` displays an example for using `plot2d` function).
- `examples` contains examples of specific topics. It is shown in appropriate subdirectories how to add new C or Fortran program to Scilab (see `addinter-tutorial`). More complex examples are given in `addinter-examples`. The directory `mex-examples` contains examples of interfaces realized by emulating the Matlab mexfiles. The directory `link-examples` illustrates the use of the `call` function which allows to call external function within Scilab.
- `macros` contains the libraries of functions which are available on-line. New libraries can easily be added (see the Makefile). This directory is divided into a number of subdirectories which contain "Toolboxes" for control, signal processing, etc... Strictly speaking Scilab is not organized in toolboxes : functions of a specific subdirectory can call functions of

other directories; so, for example, the subdirectory `signal` is not self-contained but all the functions there are devoted to signal processing.

- `man` is the directory containing the manual divided into submanuals, corresponding to the on-line help.

  To get information about an item, one should enter `help item` in Scilab or use the help window facility obtained with help button. To get information corresponding to a key-word, one should enter `apropos key-word` or use `apropos` in the help window. All the `items` and `key-words` known by the `help` and `apropos` commands are in `.html` and `whatis` files located in the `man` subdirectories.

  To add new items to the `help` and `apropos` commands the user can extend the list of directories available to the help browser by adapting the variable `%helps`.

- `routines` is a directory which contains the source code of all the numerical routines. The subdirectory `default` is important since it contains the source code of routines which are necessary to customize Scilab. In particular user's C or Fortran routines for ODE/DAE simulation or optimization can be included here (they can be also dynamically linked)

- `tests` : this directory contains evaluation programs for testing Scilab's installation on a machine. The file "demos.tst" tests all the demos.

- `intersci` contains a program which can be used to build interface programs for adding new Fortran or C primitives to Scilab. This program is executed by the `intersci` script in the `bin/intersci` directory.

1.3 Installing Scilab. System Requirements

Scilab is distributed in source code format; binaries for Windows98/NT/XP systems and several popular Unix/Linux-XWindow systems are also available. See the Scilab Web page and the contributions for specific ports. All of these binaries versions include tk/tcl interface.

The installation requirements are the following :

- for the source version: Scilab requires approximately 130Mb of disk storage to unpack and install (all sources included). You a C compiler and a Fortran compiler.

- for the binary version: the minimum for running Scilab (without sources) is about 40 Mb when decompressed.

Scilab uses a large internal stack for its calculations. This size of this stack can be reduced or enlarged by the `stacksize.` command. The default dimension of the internal stack can be adapted by modifying the variable `newstacksize` in the `scilab.star` script.

- For more information on the installation, please look at the README files.

1.4 Documentation

The documentation is made of this User's guide (Introduction to Scilab) and the Scilab on-line manual. There are also reports devoted to specific toolboxes: Scicos (graphic system builder and

simulator), Signal (Signal processing toolbox), Lmitool (interface for LMI problems), Metanet (graph and network toolbox). An FAQ is available at Scilab home page:
(`http://www.scilab.org`).

Several documents are available in French, German, Spanish, Chinese etc. See the Scilab Web page.

1.5 Scilab at a Glance. A Tutorial

1.5.1 Getting Started

After starting Scilab, you get:

```
                              ===========
                              S c i l a b
                              ===========


                             Scilab-x.y.z
                   Copyright (C) 1989-xx INRIA/ENPC


Startup execution:
  loading initial environment

 -->
```

A first contact with Scilab can be made by clicking on `Demos` with the left mouse button and clicking then on `Introduction to SCILAB` : the execution of the session is then done by entering empty lines and can be stopped with the buttons `Stop` and `Abort`.

Several libraries (see the `SCIDIR/scilab.star` file) are automatically loaded.

To give the user an idea of some of the capabilities of Scilab we will give later a sample session in Scilab.

1.5.2 Editing a command line

Before the sample session, we briefly present how to edit a command line. You can enter a command line by typing after the prompt or clicking with the mouse on a part on a window and copy it at the prompt in the Scilab window. The pointer may be moved using the directionnal arrows (←↑↓→). For Emacs customers, the usual Emacs commands are at your disposal for modifying a command (Ctrl-<chr> means hold the CONTROL key while typing the character <chr>), for example:

- Ctrl-p recall previous line
- Ctrl-n recall next line
- Ctrl-b move backward one character
- Ctrl-f move forward one character
- Delete delete previous character
- Ctrl-h delete previous character
- Ctrl-d delete one character (at cursor)
- Ctrl-a move to beginning of line
- Ctrl-e move to end of line
- Ctrl-k delete to the end of the line
- Ctrl-u cancel current line
- Ctrl-y yank the text previously deleted
- `!prev` recall the last command line which begins by `prev`
- Ctrl-c interrupt Scilab and pause after carriage return. Clicking on the Control/stop button enters a Ctrl-c.

As said before you can also cut and paste using the mouse. This way will be useful if you type your commands in an editor. Another way to "load" files containing Scilab statements is available with the `File/File Operations` button.

1.5.3 Sample Session for Beginners

We present now some simple commands. At the carriage return all the commands typed since the last prompt are interpreted.

. . . .

Give the values of 1 and 2 to the variables `a` and `A` . The semi-colon at the end of the command suppresses the display of the result. Note that Scilab is case-sensitive. Then two commands are processed and the second result is displayed because it is not followed by a semi-colon. The last command shows how to write a command on several lines by using "`...`". This sign is only needed in the on-line typing for avoiding the effect of the carriage return. The chain of characters which follow the `//` is not interpreted (it is a comment line).

. . . .

We get the list of previously defined variables `a b c A` together with the initial environment composed of the different libraries and some specific "permanent" variables.

Below is an example of an expression which mixes constants with existing variables. The result is retained in the standard default variable `ans`.

. . . .

Defining `I`, a vector of indices, `W` a random 2 x 4 matrix, and extracting submatrices from `W`. The `$` symbol stands for the last row or last column index of a matrix or vector. The colon symbol stands for "all rows" or "all columns".

. . . .

Calling a function (or primitive) with a vector argument. The response is a complex vector.

. . . .

A more complicated command which creates a polynomial.

. . . .

Definition of a structure variable.

. . . .

Definition of a polynomial matrix. The syntax for polynomial matrices is the same as for numerical matrices. Calculation of the determinant of the polynomial matrix by the `det` function.

. . . .

Definition of a matrix of rational polynomials. (The internal representation of `F` is a typed list of the form `tlist('the type',num,den)` where `num` and `den` are two matrix polynomials). Retrieving the numerator and denominator matrices of `F` by extraction operations in a typed list. Last command is the direct extraction of entry `1,2` of the numerator matrix `F.num`.

. . . .

Here we move into a new environment using the command `pause` and we obtain the new prompt `-1->` which indicates the level of the new environment (level 1). All variables that are available in the first environment are also available in the new environment. Variables created in the new environment can be returned to the original environment by using `return`. Use of `return` without an argument destroys all the variables created in the new environment before returning to the old environment. The `pause` facility is very useful for debugging purposes.

. . . .

Definition of a rational polynomial by extraction of an entry of the matrix `F` defined above. This is followed by the evaluation of the rational polynomial at the vector of complex frequency values defined by `frequencies`. The evaluation of the rational polynomial is done by the primitive `freq`. `F12.num` is the numerator polynomial and `F12.den` is the denominator polynomial of the rational polynomial `F12`. Note that the polynomial `F12.num` can be also obtained by extraction from the matrix `F` using the syntax `F.num(1,2)`. The visualization of the resulting evaluation is made by using the basic plot command `plot2d` (see Figure 1.5.3).

. . . .

The function `horner` performs a (possibly symbolic) change of variables for a polynomial (for example, here, to perform the bilinear transformation f(w(s))).

. . . .

Definition of a linear system in state-space representation. The function `syslin` defines here the continuous time (`'c'`) system `Sl` with state-space matrices (`A,B,C`). The function `ss2tf` transforms `Sl` into transfer matrix representation.

. . . .

Definition of the rational matrix `R`. `Sl` is the continuous-time linear system with (improper) transfer matrix `R`. `tf2ss` puts `Sl` in state-space representation with a polynomial `D` matrix. Note that linear systems are represented by specific typed lists (with 7 entries).

. . . .

`sl1` is the linear system in transfer matrix representation obtained by the parallel inter-connection of `Sl` and `2*Sl +eye()`. The same syntax is valid with `Sl` in state-space representation.

. . . .

On-line definition of a function, called `compen` which calculates the state space representation (`Cl`) of a linear system (`Sl`) controlled by an observer with gain `Ko` and a controller with gain `Kr`. Note that matrices are constructed in block form using other matrices.

. . . .

Call to the function `compen` defined above where the gains were calculated by a call to the primitive `ppol` which performs pole placement. The resulting `Aclosed` matrix is displayed and the placement of its poles is checked using the primitive `spec` which calculates the eigenvalues of a matrix. (The function `compen` is defined here on-line by as an example of function which receive a linear system (`Sl`) as input and returns a linear system (`Cl`) as output. In general Scilab functions are defined in files and loaded in Scilab by `exec` or by `getf`).

. . . .

Relation with the host environment.

. . . .

Definition of a column vector of character strings used for defining a C function file. The routine is compiled (needs a compiler), and a shared library is done. The libary is dynamically linked to Scilab by the `link` command, and interactively called by the function `myplus`. `myplus` passes variables from Scilab to C and conversely.

. . . .

Definition of a function which calculates a first order vector differential `f(t,y)`. This is followed by the definition of the constant `a` used in the function. The primitive `ode` then integrates the differential equation defined by the Scilab function `f(t,y)` for `y0=[1;0]` at `t=0` and where the solution is given at the time values $t = 0, .02, .04, \ldots, 20$. (Function `f` can be defined as a C or Fortran program). The result is plotted in Figure 0 where the first element of the integrated vector is plotted against the second element of this vector.

. . . .

Definition of a matrix containing character strings. By default, the operation of symbolic multiplication of two matrices of character strings is not defined in Scilab. However, the (on-line) function definition for `%cmc` defines the multiplication of matrices of character strings. The `%` which begins the function definition for `%cmc` allows the definition of an operation which did not previously exist in Scilab, and the name `cmc` means "chain multiply chain". This example is not very useful: it is simply given to show how *operations* such as `*` can be defined on complex data structures by mean of scpecific Scilab functions.

. . . .

A simple example which illustrates the passing of a function as an argument to another function. Scilab functions are objects which may be defined, loaded, or manipulated as other objects such as matrices or lists.

. . . .

```
-->quit
```

Exit from Scilab.

. . . .

A Simple Response

: Phase Plot

Chapter 2

Data Types

Scilab recognizes several data types. Scalar objects are numerical constants, booleans, polynomials, strings and rationals (quotients of polynomials). These objects in turn allow to define matrices which admit these scalars as entries. Other basic objects are lists, typed-lists and functions. The objective of this chapter is to describe the use of each of these data types.

2.1 Special Constants

Scilab provides special constants `%i`, `%pi`, `%e`, and `%eps` as primitives. The `%i` constant represents $\sqrt{-1}$, `%pi` is $\pi = 3.1415927\cdots$, `%e` is the trigonometric constant $e = 2.7182818\cdots$, and `%eps` is a constant representing the precision of the machine (`%eps` is the biggest number for which $1+$ `%eps` $= 1$). `%inf` and `%nan` stand for "Infinity" and "NotANumber" respectively. `%s` is the polynomial s=`poly(0,'s')` with symbol `s`.

(More generally, given a vector `rts`, `p=poly(rts,'x')` defines the polynomial p(x) with variable `x` and such that `roots(p) = rts`).

Finally boolean constants are `%t` and `%f` which stand for "true" and "false" respectively. Note that `%t` is the same as `1==1` and `%f` is the same as `~%t`.

These variables are considered as "predefined". They are protected, cannot be deleted and are not saved by the `save` command. It is possible for a user to have his own "predefined" variables by using the `predef` command. The best way is probably to set these special variables in his own startup file `<home dir>/.scilab`. Of course, the user can use e.g. `i=sqrt(-1)` instead of `%i`.

2.2 Constant Matrices

Scilab considers a number of data objects as matrices. Scalars and vectors are all considered as matrices. The details of the use of these objects are revealed in the following Scilab sessions.

Scalars Scalars are either real or complex numbers. The values of scalars can be assigned to variable names chosen by the user.

```
--> a=5+2*%i   //a complex number
 a         =

    5. + 2.i

--> B=-2+%i;

--> b=4-3*%i
 b  =

    4. - 3.i

--> a*b
 ans  =

    26. - 7.i

-->a*B
 ans  =

  - 12. + i
```

Note that Scilab evaluates immediately lines that end with a carriage return. Instructions that ends with a semi-colon are evaluated but are not displayed on screen.

Vectors The usual way of creating vectors is as follows, using commas (or blanks) and semi-columns:

```
--> v=[2,-3+%i,7]
 v         =

!   2.   - 3. + i       7. !

--> v'
 ans       =

!   2.        !
! - 3. - i    !
!   7.        !

--> w=[-3;-3-%i;2]
 w         =

! - 3.        !
! - 3. - i   !
!   2.       !

--> v'+w
 ans        =

! - 1.       !
! - 6. - 2.i !
!   9.       !

--> v*w
 ans        =

    18.

--> w'.*v
 ans        =

! - 6.    8. - 6.i    14. !
```

Notice that vector elements that are separated by commas (or by blanks) yield row vectors and those separated by semi-colons give column vectors. The empty matrix is `[]` ; it has zero rows and zero columns. Note also that a single quote is used for transposing a vector (one obtains the complex conjugate for complex entries). Vectors of same dimension can be added and subtracted. The scalar product of a row and column vector is demonstrated above. Element-wise multiplication (`.*`) and division (`./`) is also possible as was demonstrated.

Note with the following example the role of the position of the blank:

```
-->v=[1 +3]
 v  =

!   1.    3. !

-->w=[1 + 3]
 w  =

!   1.    3. !

-->w=[1+ 3]
 w  =

    4.

-->u=[1, + 8- 7]
 u  =
!   1.    1. !
```

Vectors of elements which increase or decrease incrementely are constructed as follows

```
--> v=5:-.5:3
 v         =

!   5.    4.5    4.    3.5    3. !
```

The resulting vector begins with the first value and ends with the third value stepping in increments of the second value. When not specified the default increment is one. A constant vector can be created using the `ones` and `zeros` facility

```
--> v=[1 5 6]
 v         =

!   1.    5.    6. !

--> ones(v)
 ans       =

!   1.    1.    1. !

--> ones(v')
 ans       =

!   1. !
!   1. !
!   1. !

--> ones(1:4)
 ans       =

!   1.    1.    1.    1. !

--> 3*ones(1:4)
 ans       =

!   3.    3.    3.    3. !

-->zeros(v)
 ans  =
!   0.     0.     0. !

-->zeros(1:5)
 ans   =

!   0.     0.     0.     0.     0. !
```

Notice that `ones` or `zeros` replace its vector argument by a vector of equivalent dimensions filled with ones or zeros.

Matrices Row elements are separated by commas or spaces and column elements by semicolons. Multiplication of matrices by scalars, vectors, or other matrices is in the usual sense. Addition and subtraction of matrices is element-wise and element-wise multiplication and division can be accomplished with the `.*` and `./` operators.

```
--> A=[2 1 4;5 -8 2]
 A         =

!   2.     1.     4. !
!   5.   - 8.     2. !

--> b=ones(2,3)
 b         =

!   1.     1.     1. !
!   1.     1.     1. !

--> A.*b
 ans       =

!   2.     1.     4. !
!   5.   - 8.     2. !

--> A*b'
 ans       =

!   7.     7. !
! - 1.   - 1. !
```

Notice that the `ones` operator with two real numbers as arguments separated by a comma creates a matrix of ones using the arguments as dimensions (same for `zeros`). Matrices can be used as elements to larger matrices. Furthermore, the dimensions of a matrix can be changed.

```
--> A=[1 2;3 4]
 A         =

!   1.     2. !
!   3.     4. !

--> B=[5 6;7 8];

--> C=[9 10;11 12];

--> D=[A,B,C]
 D         =

!   1.     2.     5.     6.     9.      10. !
!   3.     4.     7.     8.     11.     12. !

--> E=matrix(D,3,4)
 E         =

!   1.     4.     6.     11. !
!   3.     5.     8.     10. !
!   2.     7.     9.     12. !

-->F=eye(E)
 F   =

!   1.     0.     0.     0. !
!   0.     1.     0.     0. !
!   0.     0.     1.     0. !

-->G=eye(4,3)
 G   =

!   1.     0.     0. !
!   0.     1.     0. !
!   0.     0.     1. !
!   0.     0.     0. !
```

Notice that matrix `D` is created by using other matrix elements. The `matrix` primitive creates a new matrix `E` with the elements of the matrix `D` using the dimensions specified by the second two arguments. The element ordering in the matrix `D` is top to bottom and then left to right which explains the ordering of the re-arranged matrix in `E`.

The function `eye` creates an $m \times n$ matrix with 1 along the main diagonal (if the argument is a matrix `E` , m and n are the dimensions of `E`) .

Sparse constant matrices are defined through their nonzero entries (type help `sparse` for more details). Once defined, they are manipulated as full matrices.

2.3 Matrices of Character Strings

Character strings can be created by using single or double quotes. Concatenation of strings is performed by the + operation. Matrices of character strings are constructed as ordinary matrices, e.g. using brackets. An important feature of matrices of character strings is the capacity to manipulate and create functions. Furthermore, symbolic manipulation of mathematical objects can be implemented using matrices of character strings. The following illustrates some of these features.

```
--> A=['x' 'y';'z' 'w+v']
 A         =

!x  y    !
!        !
!z  w+v  !

--> At=trianfml(A)
 At        =

!z  w+v          !
!                !
!0  z*y-x*(w+v)  !

--> x=1;y=2;z=3;w=4;v=5;

--> evstr(At)
 ans       =

!   3.     9. !
!   0.   - 3. !
```

Note that in the above Scilab session the function `trianfml` performs the symbolic triangularization of the matrix `A`. The value of the resulting symbolic matrix can be obtained by using `evstr`.

The following table gives the list of some useful functions:

`ascii`	ascii code of strings
`execstr`	executes the string
`grep`	looks for a chain into a matrix
`sort, gsort`	sorting (lexicographic,...)
`part`	extract a subchain
`mmscanf`	formated read into a chain
`msprintf`	construct a chain/vector
`strindex`	location of a subchain
`string`	convert into a string
`stripblanks`	remove blank characters
`strsubst`	chain replacement
`tokens`	cuts a chain
`strcat`	catenates chains
`length`	chain length

A string matrices can be used to create new functions (for more on functions see Section 3.2). An example of automatically creating a function is illustrated in the following Scilab session where it is desired to study a polynomial of two variables `s` and `t`. Since polynomials in two independent variables are not directly supported in Scilab, we can construct a new data structure using a list (see Section 2.8). The polynomial to be studied is $(t^2 + 2t^3) - (t + t^2)s + ts^2 + s^3$.

```
-->s=poly(0,'s');t=poly(0,'t');

-->p=list(t^2+2*t^3,-t-t^2,t,1+0*t);

-->pst=makefunction(p) //pst is a function
//                  t->p (number->polynomial)
 pst       =

[p]=pst(t)

-->pst(1)
 ans       =


                2   3
    3 - 2s + s + s
```

Here the polynomial is represented by the command which puts the coefficients of the variable `s` in the list `p`. The list `p` is then processed by the function `makefunction` which makes a new function `pst`. The contents of the new function can be displayed and this function can be evaluated at values of `t`. The creation of the new function `pst` is accomplished as follows

Here the function `makefunction` takes the list `p` and creates the function `pst`. Inside of `makefunction` there is a call to another function `makestr` which makes the string which represents each term of the new two variable polynomial. The functions `addf` and `mulf` are used for adding and multiplying strings (i.e. `addf(x,y)` yields the string `x+y`). Finally, the essential command for creating the new function is the primitive deff. The deff primitive creates a function defined by two matrices of character strings. Here the function p is defined by the two character strings '[p]=newfunction(t)' and text where the string text evaluates to the polynomial in two variables.

2.4 Polynomials and Polynomial Matrices

Polynomials are easily created and manipulated in Scilab. Manipulation of polynomial matrices is essentially identical to that of constant numerical matrices. The poly primitive in Scilab can be used to specify the coefficients of a polynomial or the roots of a polynomial.

```
-->p=poly([1 2],'s')   //polynomial defined by its roots
 p         =


              2
    2 - 3s + s

-->q=poly([1 2],'s','c')  //polynomial defined by its coefficients
 q         =


    1 + 2s

-->p+q
 ans       =


             2
    3 - s + s

-->p*q
 ans       =


              2    3
    2 + s - 5s + 2s

--> q/p
 ans       =


      1 + 2s
    -----------
             2
    2 - 3s + s
```

Note that the polynomial p has the *roots* 1 and 2 whereas the polynomial q has the *coefficients* 1 and 2. It is the third argument in the poly primitive which specifies the coefficient flag option. In the case where the first argument of `poly` is a square matrix and the roots option is in effect the result is the characteristic polynomial of the matrix.

```
--> poly([1 2;3 4],'s')
 ans        =

                2
  - 2 - 5s + s
```

Polynomials can be added, subtracted, multiplied, and divided, as usual, but only between polynomials of same formal variable.

Polynomials, like real and complex constants, can be used as elements in matrices. This is a very useful feature of Scilab for systems theory.

```
-->s=poly(0,'s');

-->A=[1 s;s 1+s^2];   //Polynomial matrix

--> B=[1/s 1/(1+s);1/(1+s) 1/s^2]
 B          =

!   1           1     !
! ------      ------ !
!   s         1 + s   !
!                     !
!     1       1       !
!    ---     ---      !
!              2      !
!   1 + s     s       !
```

From the above examples it can be seen that matrices can be constructed from polynomials and rationals.

2.4.1 Rational polynomial simplification

Scilab automatically performs pole-zero simplifications when the the built-in primitive `simp` finds a common factor in the numerator and denominator of a rational polynomial `num/den`. Pole-zero simplification is a difficult problem from a numerical viewpoint and `simp` function is usually conservative. When making calculations with polynomials, it is sometimes desirable to avoid pole-zero simplifications: this is possible by switching Scilab into a "no-simplify" mode: `help simp_mode`. The function `trfmod` can also be used for simplifying specific pole-zero pairs.

2.5 Boolean Matrices

Boolean constants are `%t` and `%f`. They can be used in boolean matrices. The syntax is the same as for ordinary matrices i.e. they can be concatenated, transposed, etc...

Operations symbols used with boolean matrices or used to create boolean matrices are == and ~.

If `B` is a matrix of booleans `or(B)` and `and(B)` perform the logical `or` and `and`.

```
-->%t
 %t  =

  T

-->[1,2]==[1,3]
 ans  =

! T F !

-->[1,2]==1
 ans  =

! T F !

-->a=1:5; a(a>2)
 ans  =

!   3.    4.    5. !

-->A=[%t,%f,%t,%f,%f,%f];

-->B=[%t,%f,%t,%f,%t,%t]
 B  =

! T F T F T T !

-->A|B
 ans  =

! T F T F T T !

-->A&B
 ans  =

! T F T F F F !
```

Sparse boolean matrices are generated when, e.g., two constant sparse matrices are compared. These matrices are handled as ordinary boolean matrices.

2.6 Integer Matrices

There are 6 integer data types defined in Scilab, all these types have the same major type (see the `type` function) which is 8 and different sub-types (see the `inttype` function)

- 32 bit signed integers (sub-type 4)
- 32 bit unsigned integers (sub-type 14)
- 16 bit signed integers (sub-type 2)
- 16 bit unsigned integers (sub-type 23)
- 8 bit signed integers (sub-type 2)
- 8 bit unsigned integers (sub-type 12)

It is possible to build these integer data types from standard matrix (see 2.2) using the `int32`, `uint32`, `int16`, `uint16`, `int8`, `uint8` conversion functions

```
-->x=[0 3.2 27 135] ;

-->int32(x)
 ans  =

!0   3  27  135 !

-->int8(x)
 ans  =

!0   3  27  -121!
-->uint8(x)
 ans  =

!0   3  27  135 !
```

The same function can also convert from one sub-type to another one. The `double` function transform any of the integer type in a standard type:

```
-->y=int32([2 5 285])
 y  =
!2  5  285 !

-->uint8(y)
 ans  =

!2  5  29 !

-->double(ans)
 ans  =

!   2.     5.     29. !
```

Arithmetic and comparison operations can be applied to this type

```
-->x=int16([1 5 12])
 x  =

!1  5  12 !

-->x([1 3])
 ans  =

!1  12 !

-->x+x

 ans  =

!2  10  24 !

-->x*x'
 ans  =

 170
-->y=int16([1 7 11])
 y  =

!1  7  11 !
-->x>y
 ans  =

! F F T !
```

The operators `&`, `|` and used with these datatypes correspond to AND, OR and NOT bit-wise operations.

```
-->x=int16([1 5 12])
 x  =

!1  5  12 !

-->x|int16(2)
 ans  =

!3  7  14 !

-->int16(14)&int16(2)
 ans  =

 2
-->~uint8(2)
 ans  =

 253
```

2.7 N-dimensionnal arrays

N-dimensionnal array can be defined and handled in simple way:

```
-->M(2,2,2)=3
 M  =

(:,:,1)

!   0.     0. !
!   0.     0. !
(:,:,2)

!   0.     0. !
!   0.     3. !

-->M(:,:,1)=rand(2,2)
 M  =

(:,:,1)

!   0.9329616    0.312642  !
!   0.2146008    0.3616361 !
(:,:,2)

!   0.    0. !
!   0.    3. !

-->M(2,2,:)
 ans  =

(:,:,1)

    0.3616361
(:,:,2)

    3.
-->size(M)
 ans  =

!   2.    2.    2. !

-->size(M,3)
 ans  =

    2.
```

They can be created from a vector of data and a vector of dimension

```
-->hypermat([2 3,2],1:12)
 ans  =

(:,:,1)

!   1.    3.    5. !
!   2.    4.    6. !
(:,:,2)

!   7.    9.     11. !
!   8.    10.    12. !
```

N-dimensionnal matrices are coded as `mlists` with 2 fields :

```
-->M=hypermat([2,3,2],1:12);
-->M.dims
```

```
 ans  =

!   2.    3.    2. !
-->M.entries'
ans  =
         column  1 to 11

!   1.    2.    3.    4.    5.    6.    7.    8.    9.    10.    11. !

         column 12

!   12. !
```

2.8 Lists

Scilab has a list data type. The list is a collection of data objects not necessarily of the same type. A list can contain any of the already discussed data types (including functions) as well as other lists. Lists are useful for defining structured data objects.

There are two kinds of lists, ordinary lists and typed-lists. A list is defined by the `list` function. Here is a simple example:

```
-->L=list(1,'w',ones(2,2))  //L is a list made of 3 entries
 L  =
       L(1)

    1.
       L(2)
 w
       L(3)
 !   1.    1. !
!   1.    1. !

-->L(3);   //extracting entry 3 of list L

-->L(3)(2,2)  //entry 2,2 of matrix L(3)
 ans  =

    1.

-->L(2)=list('w',rand(2,2)) //nested list: L(2) is now a list
 L  =
       L(1)

    1.
```

```
       L(2)
        L(2)(1)
 w
        L(2)(2)
!   0.6653811    0.8497452 !
!   0.6283918    0.6857310 !
       L(3)
!   1.    1. !
!   1.    1. !

-->L(2)(2)(1,2)  //extracting entry 1,2 of entry 2 of L(2)
 ans  =

   0.8497452

-->L(2)(2)(1,2)=5; //assigning a new value to this entry.
```

Typed lists have a specific first entry. This first entry must be a character string (the type) or a vector of character string (the first component is then the type, and the following elements the names given to the entries of the list). Typed lists entries can be manipulated by using character strings (the names) as shown below.

```
-->L=tlist(['Car';'Name';'Dimensions'],'Nevada',[2,3])
 L  =
       L(1)
!Car         !
!            !
!Name        !
!            !
!Dimensions  !

       L(2)
 Nevada
       L(3)
!   2.    3. !

-->L.Name    //same as L(2)
 ans  =
 Nevada
-->L.Dimensions(1,2)=2.3

 L  =
       L(1)
```

```
!Car          !
!             !
!Name         !
!             !
!Dimensions   !

       L(2)
 Nevada
       L(3)

!   2.    2.3 !

-->L(3)(1,2)
 ans  =

    2.3

-->L(1)(1)
 ans  =

 Car
```

An important feature of typed-lists is that it is possible to define operators acting on them (overloading), i.e., it is possible to define e.g. the multiplication `L1*L2` of the two typed lists `L1` and `L2`.

2.9 Functions

Functions are collections of commands which are executed in a new environment thus isolating function variables from the original environments variables. Functions can be created and executed in a number of different ways. Furthermore, functions can pass arguments, have programming features such as conditionals and loops, and can be recursively called. Functions can be arguments to other functions and can be elements in lists. The most useful way of creating functions is by using a text editor, however, functions can be created directly in the Scilab environment using the syntax `function` or the `deff` primitive.

```
--> function [x]=foo(y)
-->    if y>0 then, x=1; else, x=-1; end
--> endfunction

--> foo(5)
 ans       =
    1.

--> foo(-3)
 ans       =

  - 1.
```

Usually functions are defined in a file using an editor and loaded into Scilab with:
`exec('filename')`.
This can be done also by clicking in the `File operation` button. This latter syntax loads the function(s) in `filename` and compiles them. The first line of `filename` must be as follows:

```
function [y1,...,yn]=macname(x1,...,xk)
```

where the `yi`'s are output variables and the `xi`'s the input variables.

For more on the use and creation of functions see Section 3.2.

2.10 Libraries

Libraries are collections of functions which can be either automatically loaded into the Scilab environment when Scilab is called, or loaded when desired by the user. Libraries are created by the `lib` command. Examples of librairies are given in the `SCIDIR/macros` directory. Note that in these directory there is an ASCII file "names" which contains the names of each function of the library, a set of `.sci` files which contains the source code of the functions and a set of `.bin` files which contains the compiled code of the functions. The Makefile invokes `scilab` for compiling the functions and generating the `.bin` files. The compiled functions of a library are automatically loaded into Scilab at their first call. To build a library the command `genlib` can be used (`help genlib`).

2.11 Objects

We conclude this chapter by noting that the function `typeof` returns the type of the various Scilab objects. The following objects are defined:

- `usual` for matrices with real or complex entries.
- `polynomial` for polynomial matrices: coefficients can be real or complex.
- `boolean` for boolean matrices.
- `character` for matrices of character strings.
- `function` for functions.
- `rational` for rational matrices (`syslin` lists)
- `state-space` for linear systems in state-space form (`syslin` lists).

- `sparse` for sparse constant matrices (real or complex)
- `boolean sparse` for sparse boolean matrices.
- `list` for ordinary lists.
- `tlist` for typed lists.
- `mlist` for matrix oriented typed lists.
- `state-space (or rational)` for syslin lists.
- `library` for library definition.

2.12 Matrix Operations

The following table gives the syntax of the basic matrix operations available in Scilab.

SYMBOL	OPERATION
`[ ]`	matrix definition, concatenation
`;`	row separator
`( )`	extraction `m=a(k)`
`( )`	insertion: `a(k)=m`
`.'`	transpose
`'`	conjugate transpose
`+`	addition
`-`	subtraction
`*`	multiplication
`\`	left division
`/`	right division
`^`	exponent
`.*`	elementwise multiplication
`.\`	elementwise left division
`./`	elementwise right division
`.^`	elementwise exponent
`.*.`	kronecker product
`./.`	kronecker right division
`.\.`	kronecker left division

2.13 Indexing

The following sample sessions shows the flexibility which is offered for extracting and inserting entries in matrices or lists. For additional details enter `help extraction` or `help insertion`.

2.13.1 Indexing in matrices

Indexing in matrices can be done by giving the indices of selected rows and columns or by boolean indices or by using the `$` symbol. Here is a sample of commands:

2.13.2 Indexing in lists

The following session illustrates how to create lists and insert/extract entries in `list` and `tlist` or `mlist`. Enter `help insertion` and `help extraction` for additinal examples. Below is a sample of commands:

2.13.3 Structs and Cells à la Matlab

The command `X=tlist(...)` or `Y=mlist(...)` creates a Scilab variable X of type `tlist` or `mlist`. The entries of `X` are obtained by the names of their fields.

For instance, if `X=tlist(['mytype','a','b'],A,B)` the command X.a returns A. If A is a matrix the command X.a(2,3) returns the entry at row 2 and column 3 of X;a, i.e. `A(2,3)`. Similarly, if `Y=mlist(['mytype','a','b'],A,B)`, we can use the syntax Y(2,3), once the extraction function `%mytype_e(varargin)` has been defined.

Also the syntax `Y(2,3)=33` can be given a meaning through the function `%s_i_mytype(varargin)`. This powerful overloading mechanism allows to define complex objects with a general indexing where indices can be fields (string) or a set of integers.

If the variable X is not defined in the Scilab workspace, then the command X.a=A creates a particular mlist which behaves as a Matlab struct. Its internal representation is similar to `X=mlist(['st','dims','a'],int32([1,1]),A)`. It is a one dimensional `struct` with one field called 'a' and the value of X.a is A. Multidimensional structs are created and manipulated in a similar way. For instance `X(2,3).a.b(2)=4` creates a 2 x 3 struct with one field a. It is represented as

```
mlist(['st','dims','a'],int32([2,3]),list([],[],[],[],[],w))
```

where w is a struct with one field 'b', and w.b is the vector [0,4]. A particular display is done for structs. Here, we have:

```
-->X(2,3)
 ans  =
 
   a: [1x1 struct]
 
-->X(2,3).a
 ans  =
 
   b: [0;4]
```

All the struct manipulations are implemented by soft coded operations i.e. Scilab overloaded functions. As structs are not basic data types some operations are slow. They have been implemented for a better Matlab compatibility.

The Matlab cells are also easily emulated. A cell is seen as a particular struct with one field called ’entries’. We just show a simple example:

```
-->X=cell(1,2)
 X  =

!{}  {}  !

-->X(2).entries=11
 X  =

!{}  !
!    !
!11  !
```

Note that Matlab uses braces `X{2}` for extracting entries from a cell.

Chapter 3

Programming

One of the most useful features of Scilab is its ability to create and use functions. This allows the development of specialized programs which can be integrated into the Scilab package in a simple and modular way through, for example, the use of libraries. In this chapter we treat the following subjects:

- Programming Tools
- Defining and Using Functions
- Definition of Operators for New Data Types
- Debbuging

Creation of libraries is discussed in a later chapter.

3.1 Programming Tools

Scilab supports a full list of programming tools including loops, conditionals, case selection, and creation of new environments. Most programming tasks should be accomplished in the environment of a function. Here we explain what programming tools are available.

3.1.1 Comparison Operators

There exist five methods for making comparisons between the values of data objects in Scilab. These comparisons are listed in the following table.

==	equal to
<	smaller than
>	greater than
<=	smaller or equal to
>=	greater or equal to
<> or ~=	not equal to

These comparison operators are used for evaluation of conditionals.

3.1.2 Loops

Two types of loops exist in Scilab: the `for` loop and the `while` loop. The `for` loop steps through a vector of indices performing each time the commands delimited by `end`.

```
--> x=1;for k=1:4,x=x*k,end
 x         =

    1.
 x         =

    2.
 x         =

    6.
 x         =

    24.
```

The `for` loop can iterate on any vector or matrix taking for values the elements of the vector or the columns of the matrix.

```
--> x=1;for k=[-1 3 0],x=x+k,end
 x         =

    0.
 x         =

    3.
 x         =

    3.
```

The `for` loop can also iterate on lists. The syntax is the same as for matrices. The index takes as values the entries of the list.

```
-->l=list(1,[1,2;3,4],'str')

-->for k=l, disp(k),end

    1.

!   1.    2. !
!   3.    4. !
```

```
 str
```

The `while` loop repeatedly performs a sequence of commands until a condition is satisfied.

```
--> x=1; while x<14,x=2*x,end
 x         =

    2.
 x         =

    4.
 x         =

    8.
 x         =

    16.
```

A `for` or `while` loop can be ended by the command `break` :

```
-->a=0;for i=1:5:100,a=a+1;if i > 10  then  break,end; end

-->a
 a  =

    3.
```

In nested loops, `break` exits from the innermost loop.

```
-->for k=1:3; for j=1:4; if k+j>4 then break;else disp(k);end;end;end

    1.

    1.

    1.

    2.

    2.

    3.
```

3.1.3 Conditionals

Two types of conditionals exist in Scilab: the `if-then-else` conditional and the `select-case` conditional. The `if-then-else` conditional evaluates an expression and if true executes the instructions between the `then` statement and the `else` statement (or `end` statement). If false the statements between the `else` and the `end` statement are executed. The `else` is not required. The `elseif` has the usual meaning and is a also a keyword recognized by the interpreter.

```
--> x=1
 x         =

    1.

--> if x>0 then,y=-x,else,y=x,end
 y         =

  - 1.

--> x=-1
 x         =

  - 1.

--> if x>0 then,y=-x,else,y=x,end
 y         =

  - 1.
```

The `select-case` conditional compares an expression to several possible expressions and performs the instructions following the first case which equals the initial expression.

```
--> x=-1
 x         =

  - 1.

--> select x,case 1,y=x+5,case -1,y=sqrt(x),end
 y         =

    i
```

It is possible to include an `else` statement for the condition where none of the cases are satisfied.

3.2 Defining and Using Functions

It is possible to define a function directly in the Scilab environment, however, the most convenient way is to create a file containing the function with a text editor. In this section we describe the structure of a function and several Scilab commands which are used almost exclusively in a function environment.

3.2.1 Function Structure

Function structure must obey the following format

```
function [y1,...,yn]=foo(x1,...,xm)
    .
    .
    .
```

where `foo` is the function name, the `xi` are the m input arguments of the function, the `yj` are the n output arguments from the function, and the three vertical dots represent the list of instructions performed by the function. An example of a function which calculates $k!$ is as follows

```
function [x]=fact(k)
  k=int(k)
  if k<1 then k=1,end
  x=1;
  for j=1:k,x=x*j;end
endfunction
```

If this function is contained in a file called `fact.sci` the function must be “loaded” into Scilab by the `exec` or `getf` command and before it can be used:

```
--> exists('fact')
 ans       =

    0.

--> exec('../macros/fact.sci',-1);

--> exists('fact')
 ans       =

    1.

--> x=fact(5)
 x         =

    120.
```

In the above Scilab session, the command `exists` indicates that `fact` is not in the environment (by the 0 answer to `exist`). The function is loaded into the environment using `exec` and now `exists` indicates that the function is there (the 1 answer). The example calculates 5!.

3.2.2 Loading Functions

Functions are usually defined in files. A file which contains a function must obey the following format

```
function [y1,...,yn]=foo(x1,...,xm)
    .
    .
    .
```

where `foo` is the function name. The `xi`'s are the input parameters and the the `yj`'s are the output parameters, and the three vertical dots represent the set of instructions performed by the function to evaluate the `yj`'s, given the `xi`'s. Inputs and ouputs parameters can be *any* Scilab object (including functions themeselves).

Functions are Scilab objects and should not be considered as files. To be used in Scilab, functions defined in files *must* be loaded by the command `getf(filename)` or `exec(filename,-1) ;` . If the file `filename` contains the function `foo`, the function `foo` can be executed only if it has been previously loaded by the command `getf(filename)`. A file may contain *several* functions. Functions can also be defined "on line" by the command using the `function/endfunction` syntax or by using the function `deff`. This is useful if one wants to define a function as the output parameter of a other function.

Collections of functions can be organized as libraries (see `lib` command). Standard Scilab librairies (linear algebra, control,...) are defined in the subdirectories of `SCIDIR/macros/`.

3.2.3 Global and Local Variables

If a variable in a function is not defined (and is not among the input parameters) then it takes the value of a variable having the same name in the calling environment. This variable however remains local in the sense that modifying it within the function does not alter the variable in the calling environment unless `resume` is used (see below). Functions can be invoked with less input or output parameters. Here is an example:

```
function [y1,y2]=f(x1,x2)
  y1=x1+x2
  y2=x1-x2
endfunction

-->[y1,y2]=f(1,1)
 y2  =
    0.
 y1  =
    2.
```

```
-->f(1,1)
 ans  =
    2.

-->f(1)
y1=x1+x2;
       !--error     4
undefined variable : x2
at line       2 of function f

-->x2=1;

-->[y1,y2]=f(1)
 y2  =
    0.
 y1  =
    2.

-->f(1)
 ans  =

    2.
```

Note that it is not possible to call a function if one of the parameter of the calling sequence is not defined:

```
function [y]=f(x1,x2)
  if x1<0 then y=x1, else y=x2;end
endfunction


-->f(-1)
 ans  =

  - 1.

-->f(-1,x2)
       !--error     4
undefined variable : x2

-->f(1)
 undefined variable : x2
at line       2 of function    f     called by :
```

```
f(1)

-->x2=3;f(1)

-->f(1)
 ans  =

    3
```

Global variable are defined by the `global` command. They can be read and modified inside functions. Enter `help global` for details.

3.2.4 Special Function Commands

Scilab has several special commands which are used almost exclusively in functions. These are the commands

- `argn`: returns the number of input and output arguments for the function
- `error`: used to suspend the operation of a function, to print an error message, and to return to the previous level of environment when an error is detected.
- `warning`,
- `pause`: temporarily suspends the operation of a function.
- `break`: forces the end of a loop
- `return` or `resume` : used to return to the calling environment and to pass local variables from the function environment to the calling environment.

The following example runs the following `foo` function which illustrates these commands.

- The function definition

  ```
  function [z]=foo(x,y)
  [out,in]=argn(0);
  if x==0 then,
       error('division by zero');
  end,
  slope=y/x;
  pause,
  z=sqrt(slope);
  s=resume(slope);
  endfunction
  ```

- The function use

```
--> z=foo(0,1)
error('division by zero');
                        !--error 10000
division by zero
at line      4 of function    foo      called by :
 z=foo(0,1)


--> z=foo(2,1)

-1-> resume
 z         =

    0.7071068

--> s
 s         =

    0.5
```

In the example, the first call to `foo` passes an argument which cannot be used in the calculation of the function. The function discontinues operation and indicates the nature of the error to the user. The second call to the function suspends operation after the calculation of `slope`. Here the user can examine values calculated inside of the function, perform plots, and, in fact perform any operations allowed in Scilab. The `-1->` prompt indicates that the current environment created by the `pause` command is the environment of the function and not that of the calling environment. Control is returned to the function by the command `return`. Operation of the function can be stopped by the command `quit` or `abort`. Finally the function terminates its calculation returning the value of `z`. Also available in the environment is the variable `s` which is a local variable from the function which is passed to the global environment.

3.3 Definition of Operations on New Data Types

It is possible to transparently define fundamental operations for new data types in Scilab (enter `help overloading` for a full description of this feature). That is, the user can give a sense to multiplication, division, addition, etc. on any two data types which exist in Scilab. As an example, two linear systems (represented by lists) can be added together to represent their parallel inter-connection or can be multiplied together to represent their series inter-connection. Scilab performs these user defined operations by searching for functions (written by the user) which follow a special naming convention described below.

The naming convention Scilab uses to recognize operators defined by the user is determined by the following conventions. The name of the user defined function is composed of four (or possibly three) fields. The first field is always the symbol `%`. The third field is one of the characters in the following table which represents the type of operation to be performed between the two data types.

Third field	
SYMBOL	OPERATION
`a`	+
`b`	: (range generator)
`c`	`[a,b]` column concatenation
`d`	`./`
`e`	`()` extraction: `m=a(k)`
`f`	`[a;b]` row concatenation
`g`	`|` logical or
`h`	`&` logical and
`i`	`()` insertion: `a(k)=m`
`j`	`.^` element wise exponent
`k`	`.*.`
`l`	\ left division
`m`	`*`
`n`	<> inequality comparison
`o`	== equality comparison
`p`	^ exponent
`q`	`.\`
`r`	/ right division
`s`	−
`t`	`'` (transpose)
`u`	`*.`
`v`	`/.`
`w`	`\.`
`x`	`.*`
`y`	`./.`
`z`	`.\.`
`0`	`.'`
`1`	<
`2`	>
`3`	<=
`4`	>=
`5`	~ (not)

The second and fourth fields represent the type of the first and second data objects, respectively, to be treated by the function and are represented by the symbols given in the following table.

Second and Fourth fields	
SYMBOL	VARIABLE TYPE
s	scalar
p	polynomial
l	list (untyped)
c	character string
b	boolean
sp	sparse
spb	boolean sparse
m	function
xxx	list (typed)

A typed list is one in which the first entry of the list is a character string where the first characters of the string are represented by the `xxx` in the above table. For example a typed list representing a linear system has the form:

```
tlist(['lss','A','B','C','D','X0','dt'],a,b,c,d,x0,'c').
```

and, thus, the `xxx` above is `lss`.

An example of the function name which multiplies two linear systems together (to represent their series inter-connection) is `%lss_m_lss`. Here the first field is %, the second field is `lss` (linear state-space), the third field is `m` “multiply” and the fourth one is `lss`. A possible user function which performs this multiplication is as follows

```
function [s]=%lss_m_lss(s1,s2)
[A1,B1,C1,D1,x1,dom1]=s1(2:7),
[A2,B2,C2,D2,x2]=s2(2:6),
B1C2=B1*C2,
s=lsslist([A1,B1C2;0*B1C2' ,A2],...
       [B1*D2;B2],[C1,D1*C2],D1*D2,[x1;x2],dom1),
endfunction
```

An example of the use of this function after having loaded it into Scilab (using for example `getf` or inserting it in a library) is illustrated in the following Scilab session

```
-->A1=[1 2;3 4];B1=[1;1];C1=[0 1;1 0];

-->A2=[1 -1;0 1];B2=[1 0;2 1];C2=[1 1];D2=[1,1];

-->s1=syslin('c',A1,B1,C1);

-->s2=syslin('c',A2,B2,C2,D2);

-->ssprint(s1)
```

```
.   | 1  2 |    | 1 |
x = | 3  4 |x + | 1 |u

    | 0  1 |
y = | 1  0 |x

-->ssprint(s2)

.   | 1 -1 |    | 1  0 |
x = | 0  1 |x + | 2  1 |u

y = | 1  1 |x + | 1  1 |u

-->s12=s1*s2;   //This is equivalent to s12=%lss_m_lss(s1,s2)

-->ssprint(s12)

    | 1  2  1  1 |    | 1  1 |
.   | 3  4  1  1 |    | 1  1 |
x = | 0  0  1 -1 |x + | 1  0 |u
    | 0  0  0  1 |    | 2  1 |

    | 0  1  0  0 |
y = | 1  0  0  0 |x
```

Notice that the use of `%lss_m_lss` is totally transparent in that the multiplication of the two lists `s1` and `s2` is performed using the usual multiplication operator `*`.

The directory `SCIDIR/macros/percent` contains all the functions (a very large number...) which perform operations on linear systems and transfer matrices. Conversions are automatically performed. For example the code for the function `%lss_m_lss` is there (note that it is much more complicated that the code given here!).

3.4 Debbuging

The simplest way to debug a Scilab function is to introduce a `pause` command in the function. When executed the function stops at this point and prompts `-1->` which indicates a different "level"; another `pause` gives `-2->` ... At the level 1 the Scilab commands are analog to a different session but the user can display all the current variables present in Scilab, which are inside or outside the function i.e. local in the function or belonging to the calling environment. The execution of the function is resumed by the command `return` or `resume` (the variables used at the upper level are cleaned). The execution of the function can be interrupted by `abort`.

It is also possible to insert breakpoints in functions. See the commands `setbpt`, `delbpt`, `disbpt`. Finally, note that it is also possible to trap errors during the execution of a function:

see the commands `errclear` and `errcatch`. Finally the experts in Scilab can use the function `debug(i)` where i=0,..,4 denotes a debugging level.

Chapter 4

Basic Primitives

This chapter briefly describes some basic primitives of Scilab. More detailed information is given in the “manual” document.

4.1 The Environment and Input/Output

In this chapter we describe the most important aspects of the environment of Scilab: how to automatically perform certain operations when entering Scilab, and how to read and write data from and to the Scilab environment.

4.1.1 The Environment

Scilab is loaded with a number of variables and primitives. The command `who` lists the variables which are available. `whos()` lists the variables which are available in a more detailled fashion.

The `who` command also indicates how many elements and variables are available for use. The user can obtain on-line help on any of the functions listed by typing `help <function-name>`.

Variables can be saved in an external binary file using `save`. Similarly, variables previously saved can be reloaded into Scilab using `load`.

Note that after the command `clear x y` the variables `x` and `y` no longer exist in the environment. The command `save` without any variable arguments saves the entire Scilab environment. Similarly, the command `clear` used without any arguments clears all of the variables, functions, and libraries in the environment.

Libraries of functions are loaded using `lib`.

The list of functions available in the library can be obtained by using `disp`.

4.1.2 Startup Commands by the User

When Scilab is called the user can automatically load into the environment functions, libraries, variables, and perform commands using the the file `.scilab` in his home directory. This is particularly useful when the user wants to run Scilab programs in the background (such as in batch mode). Another useful aspect of the `.scilab` file is when some functions or libraries are often used. In this case the commands `getf` `exec` or `load` can be used in the `.scilab` file to automatically load the desired functions and libraries whenever Scilab is invoked.

4.1.3 Input and Output

Although the commands `save` and `load` are convenient, one has much more control over the transfer of data between files and Scilab by using the Fortran like functions `read` and `write`. These two functions work similarly to the read and write commands found in Fortran. The syntax of these two commands is as follows.

```
--> x=[1 2 %pi;%e 3 4]
 x         =

!   1.          2.    3.1415927 !
!   2.7182818   3.    4.        !

--> write('x.dat',x)

--> clear x

--> xnew=read('x.dat',2,3)
 xnew      =

!   1.          2.    3.1415927 !
!   2.7182818   3.    4.        !
```

Notice that `read` specifies the number of rows and columns of the matrix `x`. Complicated formats can be specified.

The C like function `mfscanf` and `mfprintf` can be also used

```
--> x=[1 2 %pi;%e 3 4]
 x         =

!   1.          2.    3.1415927 !
!   2.7182818   3.    4.        !

--> fd=mopen('x_c.dat','w')

--> mfprintf(fd,'%f %f %f\n',x)

--> mclose(fd)

--> clear x
--> fd=mopen('x_c.dat','r')

--> xnew(1,1:3)=mfscanf(fd,'%f %f %f\n') ;

--> xnew(2,1:3)=mfscanf(fd,'%f %f %f\n')

 xnew  =

!   1.          2.    3.141593 !
!   2.718282    3.    4.       !
--> mclose(fd)
```

Here is a table of useful input-output functions:

`mprintf`	print in standard output
`mfprintf`	print in a file
`msprintf`	print in a string matrix
`mscanf`	read in standard input
`mfscanf`	read in a file
`msscanf`	read in a string matrix
`fprintfMat`	formated write a matrix into a file
`fscanfMat`	formated read of a matrix in a file
`mgetl`	read a file as a Scilab string matrix
`mputl`	write a string matrix
`mopen`	open a file
`mclose`	close a file

To manipulate binary files, the following functions are available:

`mget`	read binary data
`mput`	write binary data
`mgetstr`	print in a string matrix
`mputstr`	write a string matrix
`mgetstr`	read in a file
`mtell`	current position in a file
`mseek`	move current position
`meof`	end of file test

4.2 Help

On-line help is available either by clicking on the `help` button or by entering `help item` (where `item` is usually the name of a function or primitive). `apropos keyword` looks for `keyword` in a `whatis` file.

To add a new item or keyword is easy. Just create a `.cat` ASCII file describing the item and a `whatis` file in your directory. Then add your directory path (and a title) in the variable `%helps` (see also the README file there). You can use the standard format of the scilab manual (see the `SCIDIR/man/subdirectories` and `SCIDIR/examples/man-examples`). The Scilab LaTeX manual is automatically obtained from the manual items by a `Makefile`. See the directory `SCIDIR/man/Latex-doc`.

4.3 Useful functions

We give here a short list of useful functions and keywords that can be used as entry points in the Scilab manual. All the functions available can be obtained by entering `help`. For each manual entry the `SEE ALSO` line refers to related functions.

- Elementary functions: `sum, prod, sqrt, diag, cos, max, round, sign, fft`
- Sorting: `sort, gsort, find`
- Specific Matrices: `zeros, eye, ones, matrix`
- Linear Algebra: `det, inv, qr, svd, bdiag, spec, schur`
- Polynomials: `poly, roots, coeff, horner, clean, freq`
- Buttons, dialog: `x_choose, x_dialog, x_mdialog, getvalue, addmenu`
- GUI (Tcl-tk): `TK_EvalStr, TK_GetVar, TK_SetVar, TK_EvalFile`
- Linear systems: `syslin`
- Random numbers: `rand`, `grand`
- Programming: `function, deff, argn, for, if, end, while, select, warning, error, break, return`
- Comparison symbols: ==, >=, >, =, `&` (and),`|` (or)
- Execution of a file: `exec`
- Debugging: `pause, return, abort`
- Spline functions, interpolation: `splin2d, interp2d, smooth, splin3d`
- Character strings: `string, part, evstr, execstr, grep`
- Graphics: `plot2d, set, get, xgrid, locate, plot3d, Graphics`
- Ode solvers: `ode, dassl, dassrt, odedc`
- Optimization: `optim, quapro, linpro, lmitool, lsqrsolve`

- Interconnected dynamic systems: `scicos`
- Adding a C or Fortran routine: `link, call, addinter, ilib_for_link, ilib_build`
- Graphs, networks: `edit_graph, metanet`

4.4 Nonlinear Calculation

Scilab has several powerful non-linear primitives for simulation or optimization.

4.4.1 Nonlinear Primitives

Scilab provides several facilities for nonlinear calculations.

Numerical simulation of systems of differential equations is made by the `ode` primitive. Many solvers are available, mostly from `odepack`, for solving stiff or non-stiff systems. Implicit systems can be solved by `dassl`. It is also possible to solve systems with stopping time: integration is performed until the state is crossing a given surface. See `ode` and `dassrt` commands. There is a number of optional arguments available for solving ode's (tolerance parameters, jacobian, order of approximation, time steps etc). For ode solvers, these parameters are set by the global variable `%ODEOPTIONS`.

Minimizing non linear functions is done the `optim` function. Several algorithms (including non differentiable optimization) are available. Codes are from INRIA's `modulopt` library. Enter `help optim` for more a more detailed description.

4.4.2 Argument functions

Specific Scilab functions or C or Fortran routines can be used as an argument of some high-level primitives (such as `ode`, `optim`, `dassl`...). These fonctions are called argument functions or externals. The calling sequence of this function or routine is imposed by the high-level primitive which sets the argument of this function or routine.

For example the function `costfunc` is an argument of the `optim` primitive. Its calling sequence must be: `[f,g,ind]=costfunc(x,ind)` as imposed by the `optim` primitive. The following non-linear primitives in Scilab need argument functions or subroutines: `ode`, `optim`, `impl`, `dassl`, `intg`, `odedc`, `fsolve`. For problems where computation time is important, it is recommended to use C or Fortran subroutines. Examples of such subroutines are given in the directory `SCIDIR/routines/default`. See the README file there for more details.

When such a subroutine is written it must be linked to Scilab. This link operation can be done dynamically by the `link` command. It is also possible to introduce the code in a more permanent manner by inserting it in a specific interface in `SCIDIR/routines/default` and rebuild a new Scilab by a `make all` command in the Scilab directory.

4.5 XWindow Dialog

It may be convenient to open a specific XWindow window for entering interactively parameters inside a function or for a demo. This facility is possible thanks to e.g. the functions `x_dialog`, `x_choose`, `x_mdialog`, `x_matrix` and `x_message`. The demos which can be executed by clicking on the `demo` button provide simple examples of the use of these functions.

4.6 Tk-Tcl Dialog

An interface between Scilab and Tk-Tcl exists. A Graphic User Interface object can be created by the function `uicontrol`. Basic primitives are `TK_EvalFile`, `TK_EvalStr` and `TK_GetVar, TK_Setvar`. Examples are given by invoking the help of these functions.

Let us give a simple dialog. We pass a script to TK as a Scilab string matrix, TK opens a dialog box, and the result is returned to Scilab as a string, using `TK_GetVar`.

```
-->TK_script=["toplevel .dialog";
              "label .dialog.1 -text ""Enter your input\n here""";
              "pack .dialog.1";
              "entry .dialog.e -textvariable scivar";
              "set scivar """"";
              "pack .dialog.e"];
-->TK_EvalStr(TK_script);
-->text=TK_GetVar(scivar);
```

Chapter 5

Graphics

The graphics primitives in Scilab release 3.0 accept two graphic "styles", the "old" style which is based on a pre-processing treatment and the "new" style, used by default, which is based on an object oriented environment.

These two styles are not fully compatible. To switch to the old style, use the command `set old_style on`.

The old style mode is essentially based on two functions, `xset` and `xsetech` which act on the graphic environment. Once the graphic environment parameters are set, the graphic commands are used for plotting.

In the new style mode, once the plotting command is issued, it is possible to act on the graphic environment by changing the properties of the graphic objects made by the plot command. The graphic environment is post-processed.

Scilab graphics are based on a set a graphics functions such as `plot2d`, `plot3d` ... and an object oriented graphic environment. The properties of the graphic objects (color, thickness, ...) are manipulated with the functions `get` and `set`. The main graphic objects are the figure entity and the axes entity. We give here a description of the most useful functions illustrated by simple examples. For more sophisticated examples, it is necessary to read the on-line help pages. The graphic demos are also useful and it is a good way to start.

5.1 Function plot2d

The basic graphic function used for plotting one or several 2D curves is plot2d. The simplest graphic command is plot2d(x,y) where x and y are two vectors with same dimension. This command gives the graph of the polyline which links each point with coordinates (x(i),y(i)) with the next point with coordinates (x(i+1),y(i+1)). The vectors x et y must have at least two components. The vector x can be omitted, its default value being 1:n where n is the size of y.

Let us start with a simple example. Entering the commands

Figure 5.1: A simple graph

Figure 5.2: Two plots: superposing curves

```
-->x=linspace(0,2*%pi,100);plot2d(x,sin(x))
```

opens a graphic window and produces a graph of the sine function for 100 values of x equally spaced between 0 and 2π. The plot2d command opens a graphic window (see figure 5.1) in which the plot appears. The graph is plotted in a rectangular frame and the bottom and left sides are used for axes with ticks.

By default, plot2d does not erase the graphic window, but the plot is made in the current windows, which possibly contains older plots. Thus, if after the preceding command the following is entered :

```
t=linspace(0,4*%pi,100);plot2d(t,0.5*cos(t))
```

we get figure 5.2) in which the graph of th two functions, sine and cosine, appear with a common x axis. (note that the x-axis goes from 0 to 4π after the second plot command).

To clear the graphic window, the command xbasc() should be used or just click the Clear button of the File menu.

Using the standard menus, it is possible to zoom a part of the graph or to export the graphic window into a file or to see the 2D plot in a 3D figure. Menus can be interactively added or removed, using delmenu addmenu functions.

Parameters such as the color or the thickness of the frame can be given to the `plot2d` function. Note that the graphic parameters are given to `plot2d` with a particular syntax. For instance to select a color or a line style we can use the "style" keyword inside the plot2d command as follows:

```
-->plot2d(x,y,style=5) //5 means ``red''
```

The basic properties of a figure can be accessed by clicking on the `Edit` button of the graphic window. For instance to select a particular color (as done with style=5 before) we can use the command

```
-->plot2d(x,y)
```

and select the `Start entity picker` item of the `Edit` button, then click on the curve, select a particular color and finally select the `Stop entity picker` item.

5.2 Figure, axes, etc

Once a graphic command has been issued, the properties of the figure obtained are retrieved by the command

```
-->f=gcf()
```

Here `f` is a handle which describes the figure properties. It has many fields which can be modified to change the properties of the figure. For instance, issuing

```
-->f.figure_name='My figure'
```

will modify the `figure_name` field and change the name of the figure which appears in the current graphic window.

The handle `f` has a field called "children" which is an handle descrining the properties of the axes of the figure.

Assume now that we have plotted a graph by the `plot2d(x,y)` command want add a title to the graph obtained. The handle associated with the figure is `f=gcf()`. To access to the properties of the axes of the figure one can enter

```
-->a=f.children
```

and we see that the handle `a` has a field named "title". Now if we enter

```
-->l=a.title
```

we see that `l` is itself a handle with a field named "text". We can give a title to the plot by the command `l.text=My plot'` Of course it is possible to obtain the same result by the command

```
f.children.title.text='My plot'
```

This simple example illustrates how to manipulate graphics within Scilab.

5.3 Graphic objects properties

Help on the graphics objects can obtained by the command

```
-->help graphics_entities
```

The properties of the various graphic objects are obtained by the command

```
-->help object_properties
```

where `object_properties` is in the following table.

`agregation_properties`
`arc_properties`
`axes_properties`
`axis_properties`
`champ_properties`
`fec_properties` `figure_properties`
`grayplot_properties`
`label_properties`
`legend_properties`
`param3d_properties`
`polyline_properties`
`rectangle_properties`
`segs_properties`
`text_properties`
`title_properties`

To get a handle associated with a graphic object, one can use the functions `gcf()` (current Figure), `gca()` (current Axes), or `gce()` (current Entity). From the top figure handle, one can access the complete tree of (sub)handles by selecting the various children and parents of a given handle.

5.4 Some useful plotting functions

5.4.1 2D plots

`plot2d`	basic plot function
`plot3d`	plot surface
`param3d`	parametric plot
`(f)champ`	vector field plot
`contour`, `fcontour`, `(f)contour2d`, `contourf`, `fcontour2d`	level curve plot
`errbar`	add error bar
`Matplot`	2D plot of a matrix using colors
`histplot`	histogram
`errbar`	vertical bars
`S(f)grayplot`	smooth 2D plot of surface
`(f)grayplot`	2D plot of surface

5.4.2 3D plots

`(f)plot3d`	plot surface
`param3d(1)`	parametric plot
`contour`	level curve plot
`hist3d`	3D histogram
`genfac3d, eval3dp`	compute factes
`hist3d`	histogram
`geom3d`	3D projection
`S(f)grayplot`	smooth 2D plot of surface
`(f)grayplot`	2D plot of surface

5.4.3 Low level 2D plots

`xpoly`	polyline plot
`xpolys`	multi-polyline plot
`xrpoly`	regular polygon plot
`xsegs`	draw unconnected segments
`xfpoly(s)`	filled polygon(s)
`xrect`	rectangle plot
`xfrect`	filled rectangle plot
`xrects`	draw or fill rectangles

5.4.4 Strings, ...

`xstring, xstringl, xstringb`	string plots
`xarrows`	arrows plot
`xarc(s), xfarc, xfarcs`	arcs plot

5.4.5 Colors

`colormap`	set colormap
`getcolor`	select color
`addcolor`	add color to colormap
`graycolormap`	gray colormap
`hotcolormap`	red to yellow colormap

5.4.6 Mouse position

`xclick`	wait for a mouse click
`locate`	mouse selection of a set of points
`xgetmouse`	add color to colormap
`graycolormap`	get the current position of the mouse

5.5 Animated plot

5.5.1 pixmap mode

To run the pixmap mode, get the handle of the current figure `f=gcf();` and activate the pixmap mode `f.pixmap='on'`.

When the pixmap mode is on, the pixels of the graphic window are stored into a temporary buffer which is displayed upon request. The various steps corresponding to the creation of the image are not displayed.

An animated plot using this mode is generally obtained by a a for or while loop which produces a new image at each iteration.

When the image is ready, the screen display is realized by the show_pixmap() command.

In the animation mode, it is often necessary to set a frame with given dimensions in which the animated plotting command will take place.

To fix an appropriate frame with extremal values rect=[xmin,ymin,xmax,ymax] one can set properties of the axes (`data_bounds` field obtained by `gca()`) or alternaltively enter the command

```
plot2d([],[],rect=[xmin,ymin,xmax,ymax]).
```

Assume that y=phi(x,t) is a vector with the same dimension as x and depending on the parameter t. The animated graph of y as a function of t (for a fixed x vector), is seen by the following code :

```
f.pixmap='on';      //The data is sent into a pixmap
plot2d(x,phi(x,theta(1)));   //First y for t=theta(1)
w=gce();     //The current entity
for t=theta
 w.children.data(:,2)=phi(x,t)';  //update y
 show_pixmap(); //display the pixmap content
end
```

The role of pixmap mode is to avoid image blinking. Any modification of a graphic property implies a complete display of the new graph (including axes, titles etc). In the pixmap mode the new image is built and displayed afterwards by the show_pixmap() command.

Function xpause can be used to slow down the graphic display.

Chapter 6

Interfacing C or Fortran programs

Scilab can be easily interfaced with Fortran or C programs. This is useful to have faster code or to use specific numerical code for, e.g., the simulation or optimization of user defined systems, or specific Lapack or `netlib` modules. In fact, interfacing numerical code appears necessary in most nontrivial applications. For interfacing C or Fortran programs, it is of course necessary to link these programs with Scilab. This can be done by a dynamic (incremental) link or by creating a new executable code for Scilab. For executing a C or Fortran program linked with Scilab, its input parameters must be given specific values transferred from Scilab and its output parameters must be transformed into Scilab variables. It is also possible that a linked program is automatically executed by a high-level primitive: for instance the `ode` function can integrate the differential equation $\dot{x} = f(t, x)$ with a rhs function f defined as a C or Fortran program which is dynamically linked to Scilab (see 4.4.2).

The simplest way to call external programs is to use the `link` primitive (which dynamically links the user's program with Scilab) and then to interactively call the linked routine by `call` primitive which transmits Scilab variables (matrices or strings) to the linked program and transforms back the output parameters into Scilab variables. Note that all the parameters of the routine called by the `call` primitive must be pointers. In particular Fortran routines can be called using the `call` primitive.

Note that ode/dae solvers and non linear optimization primitives can be directly used with C or Fortran user-defined programs dynamically linked (see 6.1.1). .

An other way to add C or Fortran code to Scilab is by building an interface program. The interface program can be written by the user following the examples given in the directories `routines/examples/interface-*`. The simplest way to build an interface program is to copy/paste one of the examples given there and to adapt the code to your problem.

Examples of Matlab-like mexfunction interfaces are given in the directory `routines/examples/mexfiles`.

The interface program can also be generated by the tool `intersci`. `Intersci` builds the interface program from a `.desc` file which describes both the C or Fortran program(s) to be used and the name and parameters of the corresponding Scilab function(s).

6.1 Using dynamic link

Several simple examples of dynamic link are given in the directory `examples/link-examples`. In this section, we briefly describe how to call a dynamically linked program.

6.1.1 Dynamic link

The command `link('path/pgm.o','pgm',flag)` links the compiled program `pgm` to Scilab. Here `pgm.o` is an object file located in the `path` directory and `pgm` is an entry point (program name) in the file `pgm.o` (An object file can have several entry points: to link them, use a vector of character strings such as `['pgm1','pgm2']`).

`flag` should be set to `'C'` for a C-coded program and to `'F'` for a Fortran subroutine. (`'F'` is the default flag and can be omitted).

If the link operation is OK, scilab returns an integer `n` associated with this linked program. To undo the link enter `ulink(n)`.

The command `c_link('pgm')` returns true if `pgm` is currently linked to Scilab and false if not.
Here is a example, with the Fortran BLAS `daxpy` subroutine used in Scilab:

```
-->n=link(SCI+'/routines/blas/daxpy.o','daxpy')
linking files /usr/local/lib/scilab-2.4/routines/calelm/daxpy.o
to create a shared executable.
Linking daxpy (in fact daxpy_)
Link done
 n  =

    0.
 -->c_link('daxpy')
 ans  =

  T
-->ulink(n)

-->c_link('daxpy')
 ans  =

  F
```

For more details, enter `help link`. The `link` primitive can load a set of object files and/or a static library and/or a dynamic library.

6.1.2 Calling a dynamically linked program

The `call` function can be used to call a dynamically linked program. Consider for example the `daxpy` Fortran routine. It performs the simple vector operation `y=y+a*x` or, to be more specific,

```
y(1)=y(1)+a*x(1), y(1+incy)=y(1+incy)+a*x(1+incx),...
y(1+n*incy)=y(1+n*incy)+a*x(1+n*incx)
```

where `y` and `x` are two real vectors. The calling sequence for `daxpy` is as follows:

```
      subroutine daxpy(n,a,x,incx,y,incy)
```

The parameters of a Fortran program are pointers. The C code for the same function would be :

```
void daxpy(int *n,double *a,double *x,int *incx,double *y,int *incy)
```

To call `daxpy` from Scilab we must use a syntax as follows:

```
[y1,y2,y3,...]=call('daxpy', inputs description, ...
                     'out', outputs description)
```

Here `inputs description` is a set of parameters
`x1,p1,t1,x2,p2,t2,x3,p3,t3` ...
where `xi` is the Scilab variable (real vector or matrix) sent to `daxpy`, `pi` is the position number of this variable in the calling sequence of `daxpy` and `ti` is the type of `xi` in `daxpy` (`t='i' t='r' t='d' t='c'` stands for integer, real, double, char respectively).
`outputs description` is a set of parameters
`[r1,c1],p1,t1,[r2,c2],p2,t2,[r3,c3],p3,t3`,..
which describes each output variable. `[ri,ci]` is the 1 x 2 integer vector giving the number of rows and columns of the ith output variable `yi`. The position, `pi`, and the type, `ti`, are as for input variables (they can be omitted if a variable is both input and output).

We see that the arguments of `call` divided into four groups. The first argument `'daxpy'` is the name of the called subroutine. The argument `'out'` divides the remaining arguments into two groups. The group of arguments between `'daxpy'` and `'out'` is the list of input arguments, their positions in the call to `daxpy`, and their data type. The group of arguments at the right of `'out'` are the dimensions of the output variables, their positions in the calling sequence of `daxpy`, and their data type. The possible data types are real, integer, double and character which are indicated, respectively, by the strings `'r'`, `'i'`, `'d'` and `'c'`. Here we calculate `y=y+a*x` by a call to `daxpy` (assuming that the `link` command has been done). We have six input variables `x1=n, x2=a, x3=x, x4=incx, x5=y, x6=incy`. Variables `x1, x4` and `x6` are integers and variables `x2, x3, x5` are double. There is one output variable `y1=y` at position `p1=5`. To simplify, we assume here that `x` and `y` have the same length and we take `incx=incy=1`.

```
-->a=3;

-->x=[1,2,3,4];y=[1,1,1,1];

-->incx=1;incy=1;

-->n=length(x);  //n=4
```

```
-->y=call('daxpy',...
        n,1,'i',...
        a,2,'d',...
        x,3,'d',...
        incx,4,'i',...
        y,5,'d',...
        incy,6,'i',...
'out',...
        [1,n],5,'d');

 y  =

!   4.    7.    10.    13. !
```

(Since `y` is both input and output parameter, we could also use the simplified syntax `call(...,'out',5)` instead of `call(...,'out'[1,n],5,'d'))`.

The same example with the C function `daxpy` (from CBLAS):

```
int daxpy(int *n, double *da, double *dx, int *incx, double *dy, int *incy)
...

-->link('daxpy.o','daxpy','C')
linking files daxpy.o  to create a shared executable
Linking daxpy (in fact daxpy)
Link done
 ans  =

    1.

-->y=call('daxpy',...
        n,1,'i',...
        a,2,'d',...
        x,3,'d',...
        incx,4,'i',...
        y,5,'d',...
        incy,6,'i',...
'out',...
        [1,n],5,'d');

-->y
 y  =

!   4.    7.    10.    13. !
```

The routines which are linked to Scilab can also access internal Scilab variables: see the examples in given in the `examples/links` directory.

6.1.3 Building a dynamic library

The simple link of an object file as illustrated above may not work on some platforms. In this case, it is neccesary to build a (possibly dynamic) library containing a set of programs and to link the library with Scilab. Examples are given the directory `SCIDIR/example/link-examples-so` which are built to run in both a Linux/Unix (.so or .sl library) or Windows environment (.dll library). The library is constructed by a specific Scilab function `ilib_for_link` (which make use of libtool for Unix/ Linux libraries). Enter `-->exec ext1c.sce` at the Scilab prompt, in this directory to see the simplest C example. The script file `ext1c.sce` contains a call to `ilib_for_link` with appropriate parameters (files to be compiled, others libraries required, etc). The library is built using the environment parameters known by Scilab (compiler, linker etc). In addition a Scilab script with generic name `loader.sce` is created in the current directory. This script contains contains the code necessary to link the library with Scilab. Typically, it is a call to the `link` primitive, with an appropriate entry point (usually the name of the linked function). Note that the script file performs two tasks: building of the library and link of the library with the running Scilab. The first task is generally performed once while the second should be made in every Scilab session which needs linking the library.

6.2 Interface programs

6.2.1 Building an interface program

Many examples of interface programs are given in the directories/subdirectories `SCIDIR/examples/interface-*`

The interface programs use a set of C or Fortran routines which are given in `SCIDIR/routines/stack-c.h` and known by Scilab when this file is included in the interface program.

The simplest way to learn how to build an interface program is to look at the examples provided in the directory `interface-tutorial`.

Note that a unique interface program can be used to interface an arbitrary (but less that 99) number of functions.

6.2.2 Example

Let us consider an example given in `examples/interface-tutorial`.

We have the following C function `matmul` which performs a matrix multiplication. Only the calling sequence is important.

```
/*Matrix multiplication C=A*B, (A,B,C stored columnwise) */

#define A(i,k) a[i + k*n]
#define B(k,j) b[k + j*m]
#define C(i,j) c[i + j*n]

void matmul(a,n,m,b,l,c)
double a[],b[],c[];
int n,m,l;
{
int i,j,k; double s;
for( i=0 ; i < n; i++)
   {
     for( j=0; j < l; j++)
        {
        s = 0.;
        for( k=0; k< m; k++)
           {
    s += A(i,k)*B(k,j);
           }
         C(i,j) = s;
         }
   }
}
```

We want to have a new Scilab function (also called `matmul`) which is such that the Scilab command

```
-->C=matmul(A,B)
```

returns in `C` the matrix product `A*B` computed by the above C function. Here `A`, `B` and `C` are standard numeric Scilab matrices. Thus, the Scilab matrices `A` and `B` should be sent to the C function `matmul` and the matrix `C` should be created, filled, and sent back to Scilab.

To create the Scilab function `matmul`, we have to write a gateway function (an interface function). A gateway function is a C function which must include the header file `stack-c.h`. Fortran gatways include the file `stack.h`. These files are located in the SCIDIR/routines directory.

The following C gateway function called `intmatmul` is an example of interface for the matrix multiplication program `matmul`. See the file
`SCIDIR/examples/interface-tutorial/intmatmul.c`.

```
#include "stack-c.h"

int intmatmul(fname)
     char *fname;
{
  int l1, m1, n1, l2, m2, n2, l3;
  int minlhs=1, maxlhs=1, minrhs=2, maxrhs=2;

  /* Check number of inputs (Rhs=2) and outputs (Lhs=1) */
  CheckRhs(minrhs,maxrhs) ; CheckLhs(minlhs,maxlhs) ;

  /* Get A (#1) and B (#2) as double ("d") */
  GetRhsVar(1, "d", &m1, &n1, &l1);
  GetRhsVar(2, "d", &m2, &n2, &l2);

  /* Check dimensions    */
if (!(n1==m2)) {Scierror(999,"%s: Uncompatible dimensions\r\n",fname);
                 return 0;}

  /* Create C (#3) as double ("d") with m1 rows and n1 columns */
  CreateVar(3, "d", &m1, &n2, &l3);

  /* Call the multiplication function matmul
     inputs:stk(l1)->A, stk(l2)->B  output:stk(l3)->C    */
  matmul(stk(l1), m1, n1, stk(l2), n2, stk(l3));

  /*  Return C (3)  */
  LhsVar(1) = 3;
  return 0;
}
```

Let us now explain each step of the gateway function `intmatmul`. The gateway function must include the file `SCIDIR/routines/stack-c.h`. This is the first line of the file. The name of the routine is `intmatmul` and it admits one input parameter which is `fname`. `fname` must be declared as `char *`. The name of the gateway routine (here `intmatmul`) is arbitrary but the parameter `fname` is compulsory. The gateway routine then includes the declarations of the C variables used. In the gateway function `intmatmul` the Scilab matrices `A`, `B` and `C` are referred to as numbers, respectively `1`, `2` and `3`.

The line

```
CheckRhs(minrhs,maxrhs); CheckLhs(minlhs,maxlhs);
```

is to check that the Scilab function `matmul` is called with a correct number of RHS and LHS parameters. For instance, typing `-->matmul(A)` will give an error message made by `CheckRhs`. The function `CheckRhs` just compares the C variable `Rhs` (transmitted in the include file `stack-c.h`) with the bounds `minrhs` and `maxrhs`.
The next step is to deal with the Scilab variables `A`, `B` and `C`. In a gateway function, all the Scilab variables are referred to as integer numbers. Here, the Scilab matrices `A`, `B` and `C` are respectively numbered `1`, `2` and `3`. Each input variable of the newly created Scilab function `matmul` (i.e. `A` and `B`) should be processed by a call to `GetRhsVar`. The first two parameters of `GetRhsVar` are inputs and the last three parameters are outputs. The line

```
GetRhsVar(1, "d", &m1, &n1, &l1);
```

means that we process the RHS variable numbered `1` (i.e. `A`). The first parameter of `GetRhsVar` (here `1`) refers to the first parameter (here `A`) of the Scilab function `matmul`. This variable is a Scilab numeric matrix which should be seen ("d") as a `double` C array, since the C routine `matmul` is expecting a `double` array. The second parameter of `GetRhsVar` (here `"d"`) refers to the type (double, int, char etc) of the variable. From the call to `GetRhsVar` we know that `A` has `m1` rows and `n1` columns.
The line

```
if (n1 !=m2 )
     {Scierror(999,"%s: Uncompatible dimensions\r\n",fname);
                 return 0;}
```

is to make a return to Scilab if the matrices `A` and `B` passed to `matmul` have uncompatible dimensions. The number of columns of `A` should be equal to the number of rows of `B`.
The next step is to create the output variable `C`. This is done by

```
CreateVar(3, "d", &m1, &n2, &l3);
```

Here we create a variable numbered `3` (`1` was for `A` and `2` was for `B`). It is an array of double (`"d"`). It has `m1` rows and `n2` columns. The calling sequence of `CreateVar` is the same as the calling sequence of `GetRhsVar`, but the four first parameters of `CreateVar` are inputs.

The next step is the call to `matmul`. Remember the calling sequence :

```
void matmul(a,n,m,b,l,c)
double a[],b[],c[]; int n,m,l;
```

We must send to this function (double) pointers to the numeric data in `A`, `B` and `C`. This is done by :

```
matmul(stk(l1), m1, n1, stk(l2), n2, stk(l3));
```

Here `stk(l1)` is a double pointer to the content of the `A` matrix. The entries of the `A` matrix are stored columnwise in `stk(l1)[0]`, `stk(l1)[1]` etc. Similarly, after the call to the C function `matmul` the (double) numbers `stk(l3)[0]`, `stk(l3)[1]` are the values of the matrix product `A*B` stored columnwise as computed by `matmul`. The last parameter of the functions `GetRhsVar` and `CreateVar` is an output parameter which allow to access the data through a pointer (here the double pointers `stk(l1)`, `stk(l2)` and `stk(l3)`.

The final step is to return the result, i.e. the `C` matrix to Scilab. This is done by

```
LhsVar(1) = 3;
```

This statement means that the first LHS variable of the Scilab function `matmul` is the variable numbered `3`.

Once the gateway routine is written, it should be compiled, linked with Scilab and a script file should be executed in Scilab for loading the new function.

It is possible to build a static or a dynamic library. The static library corresponding the the example just described here is built in the directory `SCIDIR/examples/interface-tutorial` and the dynamic library is built into the directory `SCIDIR/examples/interface-tutorial-so`.

Static library

In the directory `SCIDIR/examples/interface-tutorial` just enter the `make` command in an Unix platform or in the Windows environment with the Visual C++ environment enter `nmake /f Makefile.mak`. This command produces the following file `tutorial_gateway.c` which is a C function produced by the Makefile :

```
#include "mex.h"
extern Gatefunc intview;
extern Gatefunc intmatmul;

static GenericTable Tab[]={
{(Myinterfun)sci_gateway, intview,"error msg"},
{(Myinterfun)sci_gateway, intmatmul,"error msg"},
         };

int C2F(tutorial_gateway)()
{  Rhs = Max(0, Rhs);
(*(Tab[Fin-1].f))(Tab[Fin-1].name,Tab[Fin-1].F);
  return 0;
}
```

This function is essentially the table of C functions which are dynamically linked wih Scilab.

The following file `tutorial.sce` is also produced by the Makefile :

```
scilab_functions =[...
"view";
"matmul";
           ];
auxiliary="";
files=G_make(["tutorial_gateway.o","tutorial.a", auxiliary],"void(Win)");
addinter(files,"tutorial_gateway",scilab_functions);
```

The Scilab function `addinter` makes the correspondance between the C gateway functions (such as `intmatmul`) and their names as Scilab functions.

To load the newly created function `matmul`, one has to execute this script and then the function `matmul` can be called into Scilab

```
-->exec tutorial.sce

-->A=rand(2,3);B=rand(3,3);C=matmul(A,B);    //C=A*B
```

Summing up, to build an static interface, the user has to write a gateway function such as `intmatmul`. Then he has to edit the Makefile in `SCIDIR/examples/interface-tutorial` (or a copy of it) and to put there the name of his gateway function(s) (e.g. `intmatmul.o`) in the target `CINTERFACE` and the name of the corresponding Scilab function (e.g. `matmul`) in the target `CFUNCTIONS` with the same ordering. Typing `make` produces the static library and a script file (here `tutorial.sce`) which should be executed each time the newly created function(s) are needed. Of course, it is possible to perform this operation automatically when Scilab is launched by creating a startup file `.scilab` identical to `tutorial.sce`.

Dynamic library

The directory `SCIDIR/examples/interface-tutorial-so` contains the material necessary to create a dynamic library (or a dll in the Windows environment) that can be dynamically linked with Scilab. This directory contains the following file called `builder.sce` :

```
// This is the builder.sce
// must be run from this directory

ilib_name  = "libtutorial"           // interface library name
files = ["intview.o","intmatmul.o"]  // objects files
                                     //
libs  = []                           // other libs needed for linking
table = [ "view", "intview";         // table of (scilab_name,interface-name)
          "matmul","intmatmul"];     //

// do not modify below
// ----------------------------------------------
ilib_build(ilib_name,table,files,libs)
```

The user should edit this file, which is a Scilab script, and in particular the variables `files` (a row vector of strings) anf `table` a two column matrix of strings. `files` should contain the names of all the object files (gateway functions and C functions called). Each row of `table` is a pair of two strings: the first is the name of the Scilab function, and the second the name of the gateway function. Here we have two functions `view` which has `intview` as gateway and `matmul` which has `intmatmul` as gateway. This is the example given above. After the file `builder.sce` has been edited, it should be executed in Scilab by the command

```
-->exec builder.sce
```

Scilab then generates the file `loader.sce` :

```
// generated by builder.sce
libtutorial_path=get_file_path('loader.sce');
functions=[ 'view';
            'matmul';
];
addinter(libtutorial_path+'/libtutorial.so','libtutorial',functions);
```

This file should be executed in Scilab to load the newly created function `matmul`

```
-->exec loader.sce

-->A=rand(2,3);B=rand(3,3);C=matmul(A,B);    //C=A*B
```

Summing up, to build a dynamic interface the user has to write a gateway function (such as `intmatmul`), then he has to edit the file `builder.sce` (or a copy of it) to enter the name of the Scilab function and the necessary C functions, then he has to execute the script `builder.sce`. This produce the dynamic library and the script `loader.sce`. Then each time he needs the newly created function(s), he has to execute the script `loader.sce`.

6.2.3 Functions used for building an interface

The functions used to build an interface are Fortran subroutines when the interface is written in Fortran and are coded as C macros when the interface is coded in C. An interface (gateway) routine is a standard C function or Fortran program which include the file `SCIDIR/routines/stack-c.h` (C coded gateway) or `SCIDIR/routines/stack.h` (Fortran coded gateway).

The C functions which can be used in an interface program are available as soon as `SCIDIR/routines/stack-c` is included in the source code.

The main functions are the following:

- `CheckRhs(minrhs, maxrhs)`
  `CheckLhs(minlhs, maxlhs)`

  Function `CheckRhs` is used to check that the Scilab function is called with `minrhs <= Rhs <= maxrhs`. Function `CheckLhs` is used to check that the expected return values are in the range `minlhs <= Lhs <= maxlhs`. (Usually one has `minlhs=1` since a Scilab function can be always be called with less lhs arguments than expected).

- `GetRhsVar(k,ct,&mk,&nk,&lk)`

  Note that `k` (integer) and `ct` (string) are inputs and `mk,nk` and `lk` (integers) are outputs of `GetRhsVar`. This function defines the type (`ct`) of input variable numbered `k`, i.e. the `k`th input variable in the calling sequence of the Scilab function. The pair `mk,nk` gives the dimensions (number of rows and columns) of variable numbered `k` if it is a matrix. If it is a chain `mk*nk` is its length. `lk` is the adress of variable numbered `k` in Scilab internal stack. The type of variable number `k`, `ct`, should be set to `"d", "r", "i" ,"z"` or `"c"` which stands for double, float (real), integer, double complex or character respectively. The interface should call function `GetRhsVar` for each of the rhs variables of the Scilab function with `k=1, k=2, ..., k=Rhs`. Note that if the Scilab argument doesn't match the requested type then Scilab enters an error function and returns from the interface function.

- `CreateVar(k,ct,&mk,&nk,&lk)`

  Here `k,ct,&mk,&nk` are inputs of `CreateVar` and `lk` is an output of `CreateVar`. The parameters are as above. Variable numbered `k` is created in Scilab internal stack at adress `lk`. When calling `CreateVar`, `k` must be greater than `Rhs` i.e. `k=Rhs+1, k=Rhs+2, ...`. If due to memory lack, the argument can't be created, then a Scilab error function is called and the interface function returns.

- `CreateVarFromPtr(k,ct,&mk,&nk,&lk)`

  Here `k,ct,&mk,&nk,&lk` are all inputs of `CreateVarFromPtr` and `lk` is pointer created by a call to a C function. This function is used when a C object was created inside the interfaced function and a Scilab object is to be created using a pointer to this C object.

Once the variables have been processed by `GetRhsVar` or created by `CreateVar`, they are given values by calling one or several numerical routine. The call to the numerical routine is done in such a way that each argument of the routine points to the corresponding Scilab variable. Character, integer, real, double and double complex type variables are respectively in the `cstk`, `istk`, `sstk`, `stk`, `zstk` Scilab internal stack at the adresses `lk`'s returned by `GetRhsVar` or `CreateVar`.

Then they are returned to Scilab as lhs variables. The interface should define how the lhs (output) variables are numbered. This is done by the global variable `LhsVar`. For instance

```
LhsVar(1) = 5;
LhsVar(2) = 3;
LhsVar(3) = 1;
LhsVar(4) = 2;
```

means that the Scilab function has at most 4 output parameters which are variables numbered `k= 5, k=3, k=1, k=2` respectively.

The functions `sciprint(amessage)` and `Error(k)` are used for managing messages and errors.

Other useful functions which can be used are the following.

- `GetMatrixptr("Aname", &m, &n, &lp);`

  This function reads a matrix in Scilab internal stack. `Aname` is a character string, name of a Scilab matrix. Outputs are integers `m,n` and `lp`, the entries of the matrix are ordered *columnwise*.

- `ReadString("Aname",&n,str)`

  This function reads a string in Scilab internal stack. `n` is the length of the string.

The Fortran functions have the same syntax and return logical values.

6.2.4 Examples

There are many examples of external functions interfaced with Scilab in the directories `SCIDIR/examples/interfa`
and
`SCIDIR/examples/interface-tour-so`. Examples are given in C and Fortran. The best way to build an interface is to copy one of the examples given there and to adapt the code to particular needs.

6.3 Argument functions

Some built-in nonlinear solvers, such as `ode` or `optim`, require a specific function as argument. For instance in the Scilab command `ode(x0,t0,t,fydot)`, `fydot` is the specific argument function for the `ode` primitive. This function can be a either Scilab function or an external function written in C or Fortran. In both cases, the argument function must obey a specific syntax. In the following we will consider, as running example, using the `ode` primitive with a rhs function written in Fortran. The same steps should be followed for all primitives which require a function as argument.

If the argument function is written in C or Fortran, there are two ways to call it:

- -Use dynamic link

```
-->link('myfydot.o','myfydot')
//or -->link('myfydot.o','myfydot','C')
-->ode(x0,t0,t,'myfydot')
```

- -Use the `Ex-ode.f` interface in the `routines/default` directory (and `make all` in Scilab directory). The call to the `ode` function is as above:

```
-->ode(x0,t0,t,'myfydot')
```

In this latter case, to add a new function, two files should be updated:

- The `Flist` file: Flist is list of entry points. Just add the name of your function at in the appropriate list of functions.

```
ode_list= ... myfydot
```

- The `Ex-ode.f` (or `Ex-ode-more.f`) file: this file contains the source code for argument functions. Add your function here.

Many exemples are provided in the `default` directory. More complex examples are also given. For instance it is shown how to use Scilab variables as optional parameters of `fydot`.

6.4 Mexfiles

The directories under `SCIDIR/examples/mexfiles` contain some examples of Matlab mexfiles which can be used as interfaces in the Scilab environment. The Scilab `mexlib` library emulates the most commonly used Matlab `mxfunctions` such as `mxGetM`, `mxGetPr`, `mxGetIr` etc. Not all the `mxfunctions` are available but standard mexfiles which make use of matrices (possibly sparse), structs, character strings and n-dimensional arrays can be used without any modification in the Scilab environment. If the source program is already compiled and the mexfile given as a dynamic library, it is possible to link the library with Scilab (examples are provided for Linux (.glnx extension) or Windows (.dll extension)).

6.5 Complements

6.5.1 Examining the data, function `GetData`

The functions `GetRhsVar` and `CreateVar` described above for building an interface program can be used for most programs involving matrices. Many other functions can be used for handling Scilab more complex data such as lists. See the examples in the directories examples/interfaces. It is also possible to define new objects from scratch within an interface. Let us first describe how to scan the Scilab objects. This can be done by the function `GetData`. Consider the following simple interface program:

```
#include "stack-c.h"
void intscan()
{
  int *header; double *data;
  header = GetData(1);
  printf("%i %i %i %i\n", header[0], header[1], header[2], header[3]);
  data = (double *) &header[4];
  printf("%f %f %f\n", data[0]);
}
```

This interface can be built by the following Scilab script, `scan.sce`:

```
ilib_name  = 'libintscan';
files = ['intscan.o'];
libs  = [];
table = ['scan', 'intscan'];
ilib_build(ilib_name,table,files,libs);
```

After entering `-->exec scan.sce` at Scilab prompt, a file `loader.sce` is made:

```
libintscan_path=get_file_path('loader.sce');
functions=['scan';];
addinter(libintscan_path+'/libintscan.so','libintscan',functions);
```

Entering `-->exec loader.sce` allows to call the new `scan` function. Now we can see the output of `scan(A)` for, e.g., `A=[4,2,3]`. We see that the Scilab variable `A` is represented internally by a set of 4 integers (1,1,3,0) (the "header") followed by 3 doubles (4,2,3) (the "data"). The header is made of the following: type (1 stands for "numeric matrix"), number of rows (1 since `A` is a row vector), number of columns (3 since `A` has three columns), complex type (0 since `A` is real). The data contains the entries of `A`. All entries are considered (and stored) as double. Cleary, the function can be used to examine how Scilab variables are stored internally. For instance if `B=int32(A)` we see that `B` is stored with the header (8,1,3,4) and the int data (2,3,4). All integer matrix have the type 8, and int32 receive the flag 4.

The following program shows how a matrix of strings is represented.

```
#include "stack-c.h"
void intscan()
{
  int *header;
  int k,length,start,number;
  header = (int *) GetData(1);
  /* header = [10,Mrows,Ncols,0]  ... */
  sciprint("string-type= %i, nrows= %i, ncols=%i, (reserved=) %i \n",
   header[0],header[1],header[2],header[3]);
  /* ..[1, 1+length(entry 1),., 1+ .. +length(entry M*N) ].. */
  number=header[1]*header[2];
  start=5+number;
  /*  codes entry 1,   ...,   codes entry M*N  */
  for (k=5; k<5+number; k++) {
    length=header[k]-header[k-1];
    sciprint("string %i has length %i\n", k-4, length);
    sciprint("Code: first char =%i last char = %i\n",header[start],
     header[start+length-1]);
    start=start+length;
  }
}
```

We see here that string matrices are internally represented by integers. When a string matrix is passed to an interface and the string matrix is processed by `GetRhsVar(n,'c',&M,&N,&l)` then the strings are transformed into C chars. They can be obtained by the following piece of interface:

```
#include "stack-c.h"
void intscan()
{
  int M,N,l;  char *str;
  GetRhsVar(1, "c", &M, &N, &l);
  str = (char *) cstk(l);
}
```

6.5.2 Creating new data types, function `CreateData`

Consider now the interface program:

```
#include "stack-c.h"
void intmy133()
{
  int *header;
  CreateData(1, 5*sizeof(int));
  header = GetData(1);
  header[0]=133; header[1]=1; header[2]=3; header[3]=0;
  header[4]=35;
  LhsVar(1)=1;
}
```

This interface can be built by the following Scilab script, `my133.sce`:

```
ilib_name  = 'lib133';
files = ['intmy133.o'];
libs  = [];
table = ['my133', 'intmy133'];
ilib_build(ilib_name,table,files,libs);
```

After entering `-->exec my133.sce` at Scilab prompt, a file `loader.sce` is made, in the current directory and the new scilab function `my133` is available after entering `-->exec loader.sce`. Let us call this function. We get:

```
-->X=my133()
 X  =

Undefined display for this data type
-->type(X)
 ans  =


    133.
```

A new variable has been created. Scilab knows its type `133` which is the first integer in the header, but Scilab is unable to display this variable. We can attach a generic name to the type 133, by doing the following:

```
-->typename('mytype', 133)
```

The list of current names and types is obtained by the command `[names,types]=typname()` . The command `typeof(X)` now returns `mytype`. Now we can overload a function to display `X`.

```
-->function %mytype_p(X)
-->disp('Cannot display this variable!')
```

```
-->endfunction

-->X
 X  =

Cannot display this variable!
```

Indeed, at this stage we have created a new type of variable, `133` with name `mytype`. To really display the variable, we need to write a new interface function. Let us consider the following:

```
#include "stack-c.h"
void intdisp133()
{
  int *header;
  header = GetData(1);
  header[0]=8;
  LhsVar(1)=1;
}
```

This function does nothing, except changing `header[0]` from 133 to 8. We can build a library with the two functions `my133` and `disp133` with the scilab script:

```
ilib_name  = 'lib133';
files = ['intmy133.o', 'intdisp133.o'];
libs  = [];
table = ['my133', 'intmy133';
         'disp133' ,'intdisp133'];
ilib_build(ilib_name,table,files,libs);
```

We can redefined the display function for variables tagged `"mytype"` by `typename`.

```
-->function %mytype_p(X)
-->disp(disp133(X))
-->endfunction

-->X
 X  =

35
```

When `X` is typed at the scilab prompt, the function `%mytype` is searched to display `X`, since `X` is a variable of type 133 and such variables have the generic name "mytype", set by the command `typename('mytype',133)`. The function `disp133` transforms the variable X into a standard 1 x 1 integer matrix containing the constant 35. (Note that only the type needs to be set).

6.5.3 Values or references

By default, the arguments given to the interface are passed “by value”. This means that, in the interface program, one is working with a copy of the Scilab data. Consider for instance the following gateway function:

```
#include "stack-c.h"
void inttstfct()
{ int m1,n1,l1;
  GetRhsVar(1, "d", &m1, &n1, &l1);
  stk(l1)[0] = 1;
  LhsVar(1) = 1;
}
```

for the Scilab function `X=tstfct(A)`. We assume that the variable `A` passed to `tstfct` is a standard real matrix. Note that we could use the following gateway which performs the same job.

```
#include "stack-c.h"
void inttstfct()
{ int *header; double *data;
  header = (int *) GetData(1);
  data = (double *) &header[4];
  data[0] = 1;
  LhsVar(1) = 1;
}
```

This function is installed into Scilab by executing the script

```
ilib_name  = 'libinttstfct';
files = ['inttstfct.o'];
libs  = [];
table = ['tstfct', 'inttstfct'];
ilib_build(ilib_name,table,files,libs);
```

and executing the newly created script `loader.sce`.

Assume also that `A` is the 2 x 2 matrix `A=zeros(2,2)`, and consider the Scilab command `-->B=tstfct(A)`. In the interface program, before its assignment, `stk(l1)[0]` is equal to 0, i.e. the first entry of `A`, `A(1,1)`. The output `B` will be, as expected, the Scilab matrix `[1 0;0 0]`. But the Scilab variable `A` will not be modified. This behavior is similar with the usual Scilab functions: the input parameters of the function are not modified when the function is called. It is however possible to pass the parameters by reference. This can be useful if the interfaced program do not alter the input data (the data can be used in read-only mode). Let us enter the command `-->funptr tstfct`. We get:

```
-->funptr tstfct
 ans  =

    50101.
```

which means: the function `tstfct` has number `01` in interface number `501`. It is possible to convert the gateway function into a gateway in which the variables are passed by reference. This is done as follows by the function `intppty`.

```
-->intppty(501) // adding gateway 501 to the list.

-->intppty()
 ans  =

!   6.    13.    16.    19.    21.    23.    41.    42.    501. !
```

Note that if several Scilab functions are built together in the same inerface they will all receive input parameters as reference. The behavior of `tstfct` is changed. Its argument is modified when `tstfct` is called:

```
-->A
 A  =

!   0.    0. !
!   0.    0. !
-->tstfct(A)
 ans  =

!   1.    0. !
!   0.    0. !
-->A
 A  =

!   1.    0. !
!   0.    0. !
```

Finally, we note that the function `GetRawData` gives the internal representation of the reference variable and `GetData` gives the internal representation of the value variable pointed to by the reference.

6.6 Intersci

The directory `SCIDIR/examples/intersci-examples-so` contains several examples for using `intersci` which is a tool for producing gateway routines from a descriptor file. Let us describe a simple example, `ex01`. We want to build an interface for the following C function :

```
int ext1c(n, a, b, c)
     int *n;
     double *a, *b, *c;
{
  int k;
  for (k = 0; k < *n; ++k)
      c[k] = a[k] + b[k];
  return(0);
}
```

This function just adds the two real vectors `a` and `b` with `n` entries and returns the result in `c`. We want to have in Scilab a function `c=ext1c(a,b)` which performs this operation by calling `ext1c`. For that, we provide a `.desc` file, `ex01fi.desc` :

```
ext1c a b
a  vector m
b  vector m
c vector  m

ext1c   m a b c
m integer
a double
b double
c double

out sequence c
***********************
```

This file in divided into three parts separated by a blank line. The upper part (four first lines) describes the Scilab function `c=ext1c(a,b)`. Then (next five lines) the C function is described. The last line of `ex01fi.desc` gives the name of output variables. To run `intersci` with this file as input we enter the command :

```
SCIDIR/bin/intersci-n ex01fi
```

Two files are created : `ex01fi.c` and `ex01fi_builder.sce`. The file `ex01fi.c` is the C gateway function needed for interfacing `ext1c` with Scilab. It is a gateway file built as explained above (see 6.2.2) :

```
#include "stack-c.h"
int intsext1c(fname)
   char *fname;
{
 int m1,n1,l1,mn1,m2,n2,l2,mn2,un=1,mn3,l3;
 CheckRhs(2,2);
 CheckLhs(1,1);
 /*  checking variable a */
 GetRhsVar(1,"d",&m1,&n1,&l1);
 CheckVector(1,m1,n1);
 mn1=m1*n1;
 /*  checking variable b */
 GetRhsVar(2,"d",&m2,&n2,&l2);
 CheckVector(2,m2,n2);
 mn2=m2*n2;
 /* cross variable size checking */
 CheckDimProp(1,2,m1*n1 != m2*n2);
 CreateVar(3,"d",(un=1,&un),(mn3=mn1,&mn3),&l3);/* named: c */
 C2F(ext1c)(&mn1,stk(l1),stk(l2),stk(l3));
 LhsVar(1)= 3;
 return 0;
}
```

The file `ex01fi_builder.sce` is the following :

```
// generated with intersci
ilib_name = 'libex01fi'// interface library name

table =["ext1c","intsext1c"];
ilib_build(ilib_name,table,files,libs);
```

This builder file is to be executed by Scilab after the variables `files` and `libs` have been set :

```
-->files = ['ex01fi.o' , 'ex01c.o'];
-->libs  = [] ;
-->exec ex01fi_builder.sce
```

A dynamic library is then created as well as a file `loader.sce`. Executing `loader.sce` loads the library into Scilab and executes the `addinter` command to link the library and associate the name of the function `ext1c` to it. We can then call the new function ;

```
-->exec loader.sce
-->a=[1,2,3];b=[4,5,6]; c=ext1c(a,b);
```

To use `intersci` one has to construct a `.desc` file. The keywords which describe the Scilab function and the function to be called can be found in the examples given.

List of Figures